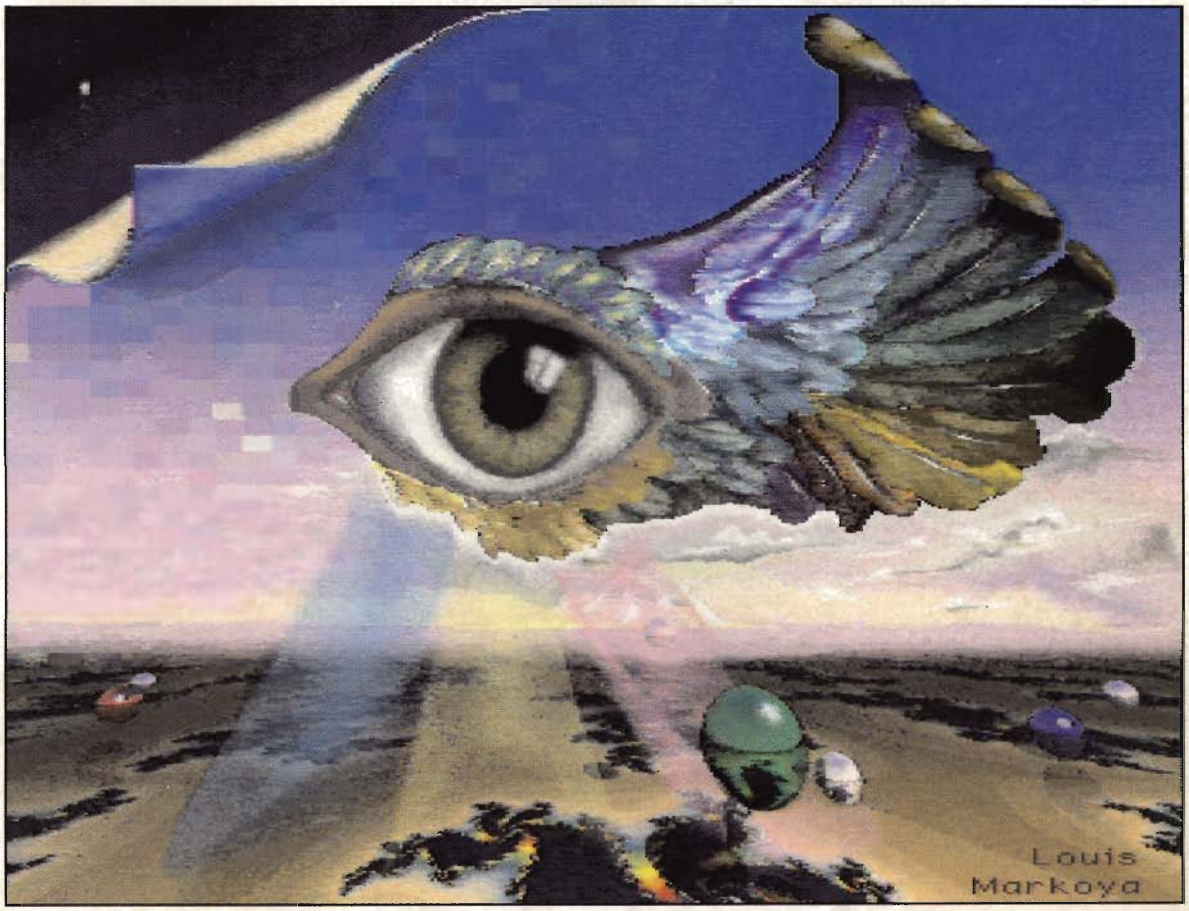
Canada $4.25
USA $3.50

FOR THE Amiga Transactor

JUNE 1988: VOLUME 1, ISSUE 2

THE MAGAZINE FOR AMIGA PROGRAMMERS

- BreakPoint - second in a series on debugging *by Victor A. Wagner*
- Preview of AmigaDOS V1.3 and summary of new commands *by Chad Innes*
- Using Workbench icons in your programs *by Rob Peck*
- All about ".info" files - a detailed examination *by Betty Clay*
- Just how random is your random number generator? *by David Wood*
- Representing objects in a CAD system *by Charles B. Blish*
- Draco: the language *by Chris Gray*
- Assembler programming with AmigaDOS *by Bob Rakosky*
- **Plus** Reviews of Photon Paint and AssemPro; regular columns by Larry Phillips, Don Curtis and Steve Ahlstrom; programming tips in *bits*; new products and more.



The Watching Angel by Louis Markoya

06
0 29074 12122

Power Tools!

For your Amiga Workbench

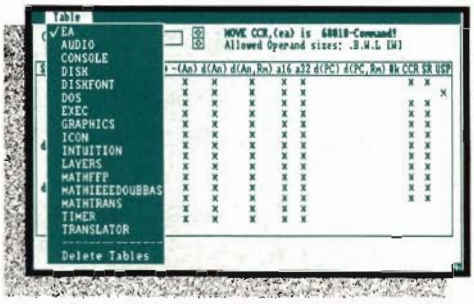
Full-screen editor

Interactive assembler

Powerful debugger

Here's a powerful development tool for the 68000 Amiga programmer. What makes AssemPro so powerful?

- Editor- a full screen editor with powerful editing features like search and replace, programmable function keys, undo typing errors, error reference file.
- Assembler- a fast two-pass macro assembler with powerful enhancements such as interactive error correction, 68010 support.
- Library- a very extensive library of operating system calls that makes it easy for you to write powerful code.
- Debugger- a complete debugging facility to power you through the testing of your programs.



Instant help

Thousands of Amiga owners already agree that AssemPro is the *power tool* they prefer. Why not find out for yourself why they use AssemPro- the professional assembly language development system. Includes 100+ page users manual .$99.95

And a guide to machine language programming

This is our practical guide to programming the Amiga in machine language. It's an Amiga-specific book, so we cover many topics in detail:

- 68000 processor- addressing modes, instruction set
- Intuition- using windows, screens, requesters, pull-down menus
- AmigaDOS- accessing the extensive libraries for input, output, disk and printer operations.
- Amiga facilities- accessing RAM, operating system, multitasking capabilities, speech and sound features

It's a great reference for all machine language programmers. and also a perfect companion for the AssemPro user. 225pp. Available June '88, $19.95. Also available is the optional program diskette containing the programs described in the book. $14.95

Abacus

Dept. TX4 • 5370 52nd Street SE
Grand Rapids, MI 49508
Phone (616)698-0330
Facsimile (616)698-0325 • Telex 709-101

Call or write today for your free catalog of our new Amiga software and books or for the name of your nearest dealer. You can order direct by phone with your VISA, American Express, or Mastercard or send your completed order blank. 30-day money back guarantee on software. Dealer inquiries welcome- over 2400 dealers nationwide.

Yes! Please rush me ____ copies of **AssemPro Amiga** $99.95 _____
Yes! Please rush me ____ copies of **Amiga Machine Language** $19.95 _____
Yes! Please rush me ____ optional program diskettes $14.95 _____
In U.S. add $4.00 shipping & handling (outside U.S. add $12.00 per item) _____
Michigan residents include 4% sales tax _____
Total amount (U.S. funds) _____

Payment: MC VISA AmEX Check Money order
Card# _____ Exp. _____
Name _____
Address _____
City _____ State _____ Zip _____
Phone(_____) _____

___ Please send me a free catalog of your complete line of Amiga software and books.

The fastest

Modula-2

Software Development System for

AMIGA

$ 249

Demonstration disk **$ 10**

Modula-2 is the successor of Pascal. Both languages were invented by Professor Niklaus Wirth. Modula-2 is an all purpose language suited for educational, home and professional use. Modula-2 leads to good programming technique, easy readability and maintainable programs. In fact, a lot of schools and universities have chosen Modula-2 for their programming courses.

M2Amiga is a software development system for Modula-2 based on an extremly fast single pass compiler. It is fully integrated into Amiga Workbench and is easy to use from CLI or Workbench. It runs on all Amiga models with a minimum configuration of 512k RAM and one disk drive.

M2Amiga has been developed specially for the Amiga and therefore optimally supports the features of this unique computer.

- Single pass compiler produces optimized native MC68000 code from standard ASCII files. Full support of latest edition of Modula-2.
- Fastest library calls possible on the Amiga. No intermediate code needed for library calls and parameter passing.
- Supports FFP, 32 bit IEEE and 64 bit IEEE reals formats, you can even mix all three formats within the same program.
- Easy access to MC68000 registers, allows the insertion of inline code..
- Full interface definition to the Amiga's operating system following the official definitions.
- The best runtime system currently available on the Amiga. Opens and closes all used libraries, captures all GURUs and releases all used resources after program crash or termination. Routines for System requesters enable user interface consistent with the Amiga's Intuition.
- The emacs based editor places the cursor on compilation errors and explains them in full text.
- In just a few seconds the linker produces stand-alone applications that can be run from CLI or Workbench. Stand-alone applications do not contain any useless overhead – they are compact and fast.
- All programs can produce icons for their output files to allow effective use of the intuitive Workbench environment. Experience the new feeling of program development by selecting and double clicking icons!
- Also included are a project management tool and other utilities.
- Besides the handy manual, an introductory guide to Modula-2 is included. This documentation allows both the beginner and the advanced user to find any information needed quickly and easily.
- Demo programs show the use of various features of the Amiga.
- For professional program development special utilities are available.
  + Source Level Debugger, the new way of debugging your programs. Inspect all your variables (displayed in their own format), zoom into structures as records or arrays or follow pointer chains by clicking the mouse buttons
  + Object file converter, converts standard Amiga object files into M2Amiga format. This allows linking of routines written in other languages.
  + Linker for devices or libraries. Add power to the Amiga by supplying new libraries and devices. Easy and intuitive programming instead of fooling around with Assembler and Amiga linker are possible now!
  + Modula-2 Amiga Programming System Environment

We have Modula-2 Compilers for HP-UX, IBM/370, PCs (Taylor, M2SDS, JPI), OS-9 and Sun, tomorrow even more!

Texas residents add 8% sales tax. International Orders add $ 20 shipping

The Modula-2 people: INTERFACE TECHNOLOGIES

3336 Richmond, Suite 323, Houston, TX 77098-9990 (713) 523 8422

Dealer inquiries welcome

2U

A. + L. Meier-Vogt
Im Späten 23
CH-8906 Bonstetten/ZH
Switzerland
Tel. (41) (1) 700 30 37

Transactor for the Amiga is now offering for sale the programmers' text editor/word processor regarded by many as the best program of its kind...

Uedit

by Rick Stiles

This is a small ad, and there isn't room to do more than *begin* to list Uedit's features. So let's see how far we can get...

Uedit supports up to 100 text buffers... you can mark, delete and recall two different kinds of highlighted region in each one... you can also delete and recall words and lines at the touch of a key... Uedit is fully configurable - key assignments can be quickly and conveniently modified to your taste... command sequences can be attached to keys... new commands can be written in the fully documented command language to perform virtually any editing task you can imagine... Printers are fully supported - Uedit even has a built-in spooler!... Use true tabs, or emulate tabs with spaces. In either mode, you can set up 5 fully configurable tab tables, which are saved as part of your personal editing environment... A powerful 'learn' feature takes most of the work out of repetitious command sequences... Personalized menus and gadgets can be set up in your configuration file.... the amount of text you can edit is limited only by the amount of memory in your machine... in addition to normal cut and paste, columnar cut and paste operations are supported... A new 'split window' feature lets you look at multiple files simultaneously, or look at different parts of the same file... Uedit is so configurable that emulations of most other popular editors can be and have been written in Uedit's command language - including Wordstar, Vi, MicroEmacs and more!... Uedit's 'tiny window' mode conserves Chip RAM...

Out of space! See the card at the centre of this magazine for ordering details!

saf-T-net

The ULTIMATE Hard Disk Backup Utility!

- o Intuition-based, runs from Workbench or CLI.
- o Diskettes formatted on an as-needed basis.
- o Full or Incremental backup utilizing Archive bit.
- o Selective and full restore capability.
- o Wild-card patterns used for inclusion/exclusion.
- o Fast - approx. 3 minutes (or less) per diskette.
- o Preservation of Date & time, Comments, and status.
- o Sophisticated error recovery.

$49.95.

R.S.N. Software, Inc.
4122 S. Parker Road
Aurora, CO 80014
(303) 690-0102

Dealer Inquiries Welcome.

Volume 1, Issue 2

Publisher
Richard Evers

Editors
Nick Sullivan
Chris Zamara

Assistant Editor
Malcolm O'Brien

Editorial Assistant
Moya Drummond

Customer Service
Renanne Turner

Contributing Writers

Steve Ahlstrom
Scott Ballantyne
Charles B. Blish
Jim Butterfield
Betty Clay
Don Curtis
Matt Dillon
Andy Finkel
Chad Innes
Chris Gray
Perry Kivolowitz
Rico Mariani
Bryce Nesbitt
Rob Peck
Larry Phillips
Bob Rakosky
John Toebes
Victor A. Wagner
David Wood

Cover Artist
Louis Markoya

for the AMIGA Transactor
The Magazine for Amiga Programmers

BreakPoint, Part 2 — 18
by Victor A. Wagner
The second instalment in Vic's debugging series reveals one of the simplest and most powerful tools - the PRINT statement!

Preview of AmigaDOS V1.3 — 22
by Chad Innes
A rundown of the new features in the upcoming Workbench release - there's some nice stuff in here!

AmigaDOS V1.3 Manual Update — 24
by Chad Innes
In 1.3, there are some new CLI commands, and some existing ones have been changed. Here's the documentation to bring you up to date.

Workbench Musings — 26
by Rob Peck
What the Workbench is from the system's point of view, and how to use Icons in your C programs, with sample programs.

Info about .Info — 32
by Betty Clay
An excursion through the icon files on the Workbench disk that reveals what's in there and why.

Random, Or Just Confused? — 38
by David Wood
We take our random number generators for granted, but how do they stand up to statistical analysis? This article gives you the mathematical theory you need to find out.

About Draco — 47
by Chris Gray
Draco is a public domain language available for the Amiga. Chris Gray, its author, introduces the language and compares it with C and Pascal.

Draco - the Language and Compiler — 50
by Chris Gray
Draco's insides are revealed in this technical article for the compiler fan. Chris talks about Draco design decisions, philosphy, and how it's put together.

Representing Objects in a CAD System — 56
by Charles B. Blish
The author of the PCLO programs for the Amiga explains an approach for fast CAD program algorithms.

Programming With AmigaDOS — 59
by Bob Rakosky
The development of an "Update" command for DOS in assembler is a springboard for an explanation about programming with DOS and the new ARP library. With assembler source listing.

Departments and Columns

Letters 4

The ViewPort 6

by Larry Phillips

User interface sins and follies - what makes a program easy and enjoyable to use.

Bits 8

Easy rule for Minterms
Tiny-type
Race conditions
An Intui-bug
Double buffering under Intuition
Reading Workbench arguments
Disk Rescue
Single drive disk copy

Amiga Dispatches 12

by Don Curtis

More about the SideCar and A2000/BridgeBoard Combination. Also clones, emulators and rumours.

ACCESS - The best of non-commercial software 15

by Steve Ahlstrom

Two shareware programs investigated in depth: The CLI Wizard, and File PAKer 1.0

News LINK 77

Oops! 80

A bug fix for "Cycle Warriors"

Reviews

AssemPro 72

An integrated assembler development system with plenty of features, and a few problems.

Photon Paint 74

by Louis Markoya

The new HAM-mode paint package by MicroIllusions breaks new ground for Amiga graphics tools.

About the cover: *The Watching Angel* by Louis Markoya. His explanation follows:

To start, a wing was digitized using Digi-View in Hi-Res Black and White mode. This was brought into Photon Paint and separate feathers were cut, rotated, bent, resized and pasted to form the wing. The eye was drawn freehand using the ellipse and fill tools, employing various modes of shading. The feathers were individually colourized by first tinting and then shading, using the freeform fill and blending modes. Extensive use of the magnification window was used to touch up the wing and eye, like adding the highlights to the feather shafts. Once finished, the image was saved as a brush to be placed later in the picture.

For the background I first defined a Julia Set Fractal using MandFXP. This was mapped as terrain with Silver, the ray tracing and animation package from Impulse. I then took this image into Photon Paint and touched up the ground and spheres, shaded the sky and added the clouds. To give the picture a more computer-generated look I used the Pixelization mode of Photon starting on the ground at 2 pixel squares and progressing to the maximum 9 in the sky. The previously-saved brush was called up, and I remapped the picture palette to match it. Once placed, fine details were done in the Magnification mode. The transparent rays were added using the blend mode. To give a more striking sense of depth, the corner of the picture was drawn folded back using the curve and fill tools. This picture is dedicated to my wife Helen.

TransAmi's
phone number is:
(416) 764-5273
Mondays, Wednesdays
and Fridays ONLY

FAX: (416) 764-9262

Transactor for the Amiga is published bimonthly by Transactor Publishing Inc., 85-10 West Wilmot Street, Richmond Hill, Ontario, L4B 1K7. ISSN# 0838-0163. Canadian Second Class Mail Registration No. 7690, Gateway-Mississauga, Ont. US Second Class mail permit pending at Buffalo, NY. USPS Postmasters: send address changes to: Transactor, PO Box 338, Station C, Buffalo, NY, 14209.

Transactor Publishing Inc. is in no way connected with Commodore Business Machines Ltd. or Commodore Incorporated. Commodore and Commodore product names are registered trademarks of Commodore Inc.

Subscriptions:

Canada $19 Cdn.
USA $15 US
All others $21 US
Air Mail (Overseas only) $40 US

Send all subscriptions to: ransactor Publishing Inc., Subscriptions Department, 85-10 West Wilmot Street, Richmond Hill, Ontario, Canada, L4B 1K7, (416) 764-5273. Note: Subscriptions are handled at this address ONLY. Subscriptions sent to our Buffalo address (above) will be forwarded to our Richmond Hill HQ. For best results, use the postage paid card at the centre of the magazine.

Quantity Orders: In Canada: Ingram Software Ltd., 141 Adesso Drive, Concord, Ontario, L4K 2W7, (416) 738-1700. In the USA: IPD (International Periodical Distributors), 11760-B Sorrento Valley Road, San Diego, California, 92121, (619) 481-5928; ask for Dave Bruescher. Quantity orders/enquiries are also welcome from computer/software distributors in the UK, Europe and Scandinavia. Please contact: T.G. Hamilton (W/S) Ltd., Tel: 021-742-5359; Fax: 021-742-2190; or contact Transactor (UK) direct at Unit 2, Langdale Grove, Bingham, Notts. NG13 8SR, telephone 0949-39380.

Editorial contributions are always welcome. Only original, previously unpublished material will be considered. Program listings and articles, including BITS submissions, of more than a few lines, should be provided on disk. Preferred format is Amiga 3.5-inch diskettes with ASCII text files. Manuscripts should be typewritten, double-spaced, with special characters or formats clearly marked. Photos should be glossy black and white prints. Illustrations should be on white paper with black ink only. IFF picture files or Aegis DrawPlus structured graphics files are preferred to hardcopy illustrations when possible.

All material accepted becomes the property of Transactor Publishing Inc., except by special arrangement. All material is copyright by Transactor Publishing Inc. Reproduction in any form without permission is in violation of applicable laws. Write to the Richmond Hill address for a writer's guide.

The opinions expressed in contributed articles are not necessarily those of Transactor Publishing Inc. Although accuracy is a major objective, Transactor Publishing Inc. cannot assume liability for errors in articles or programs. Programs listed in Transactor for the Amiga, and/or appearing on Transactor for the Amiga disks, are copyright by Transactor Publishing Inc. and may not be duplicated or distributed without permission.

Production
In-house with Amiga 2000 and Professional Page

Final output by Vellum Print & Graphic Services, Inc., Toronto

Printing
Printed in Canada by Maclean Hunter Printing

Letters

Device Handler Wanted: First, I want to congratulate you on your new Amiga Magazine; the announcement is the best news I've heard in a long time.

I am currently writing a BBS on my Amiga 500 in Lattice C. I need to create my own device that will send output to both the BBS window and the modem. It would merge all input from both the modem and the keyboard into a continuous stream. Let's say I call the device BBS: - if the user were to post a Bulletin, my program would run the Edit program as follows:

Edit >BBS: <BBS: filename

I can write a program that would do these functions, but I do not know how to make the Amiga treat it as a device so that I can open a file to it or use CLI redirection as above. Thank you very much.

Howard C. Cochran, Lilburn, Georgia

What you need is a custom device handler, which is not a simple thing to implement. An example of how to write one would be too long to list here, but some examples that have been written are in the public domain. A very complete example by Matt Dillon called 'DosDev' shows how to make a RAM-disk and is an excellent basis for any custom DOS device. The source code and documentation for DosDev can be found on fish disk #113. The resulting handler is put into DEVS: (usually the "devs" directory on the boot disk) and the device is added to the system by creating a "MountList" and using the standard "Mount" command. We haven't had the experience of a project like this, but given the basic device handler code, it doesn't sound like it would be too difficult to implement.

Incidentally, it sounds like an interesting approach to a BBS program - we wish you success with it.

A2000 Compatibility Problems: On the subject of A2000 compatibility - not all programs that worked on the 1000 with Kickstart 1.2 will do so on the new A2000. So far I have found Arctic Fox, Halley Project, Archon, and Diablo refuse to comply. Diablo was only purchased last night so that was really a big disappointment (I've written to Classic Image to inform them of the problem already). I'm sure there will be other programs that will have similar difficulties with autoloaders. Just thought your readers would like to know what to expect.

Joyce Hancock, St. Louis, Missouri

Unfortunately, these programs and some others don't get along with early Amiga 2000s. The fault is not with Commodore for releasing an incompatible machine, however; as far as we know, the operating system is identical on all current Amigas. Compatibility problems such as these are most often caused by programs that deal with Amiga hardware directly instead of using the system mechanisms provided for doing so. In the case of the 2000s, the keyboard timing was changed. This is a reasonable hardware difference that should be of no consequence to an application, but when a program plays against the rules and uses the hardware directly, all bets are off. The Amiga documentation is quite clear about what your program should and shouldn't do, so no one can blame Commodore for making a low-level change such as this. There's no guarantee that there won't be more such changes in future machines, and if a program breaks because of one, it means it was written incorrectly.

That information doesn't help you, the user, of course. From your point of view, the fact is that part of your software collection has been made useless because you upgraded your machine. In this case, your only hope is that the software manufacturers will provide free upgrades; they will have to make the upgrade anyway to service new users of the 2000.

Hopefully, this incident will teach software developers some things about compatibility, and the problem won't repeat itself with future Amiga hardware configurations.

C vs. PASCAL/MODULA 2: Congratulations on an excellent premiere issue of *Transactor for the Amiga*. It was really interesting to read about CAOS and the arp.library. Also, Chris Zamara's review of Lattice C, V4.0 was the best review of a compiler I've ever read. Betty Clay's article on fonts was one of the best written technical articles I've read. It's very apparent that she is a teacher!

I am an experienced programmer, having worked for over 10 years in PASCAL, FORTRAN and assembly. I have a couple of questions which I've never seen anyone address, so I am writing to see if you could do an article on them.

I've never worked with C but from what I've read and heard, it doesn't sound like a language I would be comfortable with. It is too terse and too difficult for someone else to understand. It also allows you to do tricky things without explicitly indicating that they are tricky. I much prefer Pascal where all variables must be typed and the compiler catches all non-standard instructions. So, as I stand on the verge of buying some development tools and writing some software for the Amiga, my questions are the following:

1. What would I give up by programming the Amiga in Pascal (or Modula-2): speed, compact code, access to system routines?

2. Scott Ballantyne's article on arp.library pointed out that BCPL routines have trouble interfacing with the system. Will Pascal and Modula-2 have the same problem?
Well, keep up the good work. I'm looking forward to the next issue of *Transactor for the Amiga*.

Carl Nagy, Orange, California

Currently, the best alternative high level language to C on the Amiga is Modula 2. There are currently at least three commercial Modula packages available. For the most part, you will have to give up very little by going to Modula instead of C, and in terms of efficiency of generated code, the languages are very close overall. If you like M2 better than C, we would suggest you go with it.

As far as access to system routines is concerned, these implementations all let you access all the system library functions, and the kinds of problems that BCPL programs have do not exist. C is still more convenient in many ways (when it comes to initializing data structures, for example), but in general just about anything you can do in C can be accomplished with Modula-2.

Benchmark Modula-2 was reviewed in Transactor, Volume 8 Issue 4; look for a review of M2Amiga in an upcoming issue of TransAmi.

C and Header Files: I am impressed with the first issue of *Transactor for the Amiga* and I have mailed the subscription card. After reading some of the articles which give C language examples I would like to make the following observation. Many of the people who can benefit from your magazine do not have a C compiler. I am such a person. Even though I use C professionally, I prefer Assembly for a home computer. The C language is not easy to read unless the programmer formats his code carefully and is very thorough with his comments. The situation is made worse by the common use of header files or other "include" files.

I am concerned that your writers are falling into the trap that so many C language programmers do not seem to be able to avoid. Some of the C code in your first issue cannot be fully understood without reference to the header files. Clearly one doesn't have to have a C compiler to gain valuable information from C language programming examples, and your magazine will appeal to a wider readership if sample programs are self contained and fully intelligible. Therefore, for the purpose of communicating with your readers, I urge your writers to explicitly include at the head of their code all relevant definitions from the header files. In most example programs this will add only a few lines. Perhaps these lines could be in the form of a comment so that it would be clear that they need not be compiled.

Another solution, although less effective, would be to publish the header files. I have not seen the header files for the Amiga system, but I should think they would be filled with useful information if they are anything like Unix header files.

Richard Williams, Columbia, Maryland

We realize that not everyone wishes to program in C, so we will continue to print Assembler articles and programs.

You make a good point about interpreting C code that relies heavily on symbols defined in header files. The header file definitions, though, are part of programming on the Amiga, and you eventually get to know the commonly used defines and structures after writing a few programs. Listing the symbols used from the header files at the top of the program might make it clearer for non-Amiga programmers, but the extra work that entails for the programmer takes away, to a certain extent, the convenience of using the header files in the first place.

Listing the header files in the magazine is a problem for two reasons: they are far too long, and they are Copyright by Commodore-Amiga Inc. and may not be published, distributed, etc. without their permission. And while some of the structure definitions in the header files would be of interest, the values of constants, flags, and other "magic numbers" is useless information if you're just reading the code to understand it and don't have a C compiler.

If you want the header files themselves, you should be able to get them from Commodore - this is a good idea if you want to become acquainted with the system structures, since the files are well commented and organized. These will also be helpful to you if you have a public domain 68000 assembler and want to use Intuition and the rest of the system with your assembler programs.

We will keep your suggestions in mind and consider programmers without C compilers in the future, especially for programs that don't need a lot of header file information. In the long run though, the header files will always be with us, and programs will continue to make extensive use of them.

Bridgeboard Combos: Congratulations on the *Transactor for the Amiga*. It is so refreshing to read some real Amiga information instead of game reviews and Amiga show articles. The experience of Don Curtis (Amiga Dispatches) in setting up his A2000 and A2088 Bridgeboard was very helpful. Please print more articles concerning this marriage.

I would especially like to see any information about how to install a hard drive and an interface with the A2000 / Bridgeboard combination. I cannot locate this information anywhere.

Thank you for a fine publication.

Donald J. Gasior, Sterling Heights, Michigan

The View Port

User interface sins and follies

by Larry Phillips

Larry Phillips is an Amiga Hardware and software hacker from Vancouver, British Columbia, and a Sysop on CompuServe's AmigaForum.

A friend of mine once said, "The reason I like the Amiga so much is that so many talented people like it too." I tend to agree with him, especially when I look at the number and quality of programs available for the machine. This includes the Fred Fish and Amicus collections of public domain and shareware programs, as well as the many commercial offerings, and probably more that haven't been widely distributed for one reason or another.

While there are dogs among them, the public domain offerings are, for the most part, of reasonable quality, with a good percentage of true gems. Considering that the authors of these do not ask for any compensation, the level of software in this category is more than adequate. Shareware, the next rung up the ladder, tends to be of the same or better quality, though there are those programs whose authors, in asking for money, are either overly optimistic, or are blissfully unaware that their offerings are either poorly done or offered as PD by someone else.

It isn't really surprising that there are a lot of good programs out there, since the Amiga is one of those machines that appeal to many different types of people, whether they be artists, programmers, musicians, or just plain 'techno-junkies'. I guess what does surprise me is the number of commercial programs that are just not up to snuff, complete with hostile or non-Amiga user interfaces, lack of features, non-cooperation with other tasks in the machine, or, worst of all, flagrant bugs demanding an upgrade that never seems to come. It seems somehow acceptable to have flaws in a program that costs nothing, or that you have a chance to try before you buy, but the same flaws in a commercial product are a lot harder to take.

The other day, I was perusing some old programs, dating back to late 1985, and ran across a game program written in ABasiC. ABasiC, for those who haven't used it, was the BASIC provided with the early Amigas. I loaded up ABasiC to run the game, and spent some time playing with ABasiC itself.

I can remember my reaction to ABasiC back then, thankful that I had a language I could use right away, but disappointed in a number of 'features', such as the line-oriented editor. At that time, rumours of a new BASIC abounded, and all Amiga owners looked forward to its arrival, hoping against hope that maybe, just maybe, Microsoft would produce a good product for the first time in their history.

When it arrived, I was again disappointed, and again it was the user interface that was the culprit. The language itself, though buggy, provided a number of extensions to standard BASIC, and reading the manual was enough to make me think that perhaps BASIC on the Amiga might be usable. Unfortunately, it isn't (though some would disagree), because of the hostile environment one is forced to interact with. Slow windows, and AmigaBasic's insistence on redrawing them at the drop of a hat, the interpreter's annoying habit of pointing out syntax errors in a program even when you are trying to do something completely unrelated in the command window, and the world's second most unfriendly editor, all combine to completely alienate me. When I saw AmigaBasic, I thought that perhaps ABasiC wasn't so bad after all. I can stand to have AmigaBasic loaded up only for short periods of time, and tend to avoid doing so except when I need to check out a program written in BASIC. If it had been better in the user interface department, at least I could have used it as a language for quick prototyping or one-time programs.

This state of affairs made me stop and think about user interfaces, and how they affect the user's perception of a program. A bad program, or a buggy one, is of no use at all, and no amount of effort spent on the user interface will do anything for it, but a mediocre program could be perceived as good, simply by virtue of being friendly and easy to use. A program that one would classify as 'great' probably already has a good user interface, and that is likely to be at least part of the reason for its success.

AmigaBasic is a perfect example of a mediocre program that could have been made better by the application of a little thought. The 'Syntax Error' message does not need a requester, with the resulting inconvenience to the user of having to wait for it to render, and then to have to use the mouse to cancel it. To me, having to move my hand off the keyboard to

use the mouse is unacceptable. It would be a different matter had I been using the mouse already, but since I am generally programming when I see that requester, I have both hands on the keyboard.

There are things for which the mouse is appropriate, and things for which the mouse seems forced and unnatural, and programmers should always bear in mind that the keyboard is for many purposes at least as good an input device as the mouse. Where possible, give users a choice - they will appreciate it. The AmigaBasic editor is the epitome of unfriendliness, slow and unforgiving, requiring the patience of a saint should you hold down a cursor key too long, and having only a few redeeming virtues, such as auto-indent and auto-capitalization of reserved keywords.

Unlike some other machines that use a mouse/icon based interface, the Amiga does not attempt to force software developers into having consistent user interfaces, but allows them the freedom to innovate and create. This has both good and bad effects on the available programs. On the good side, we do have beautiful and imaginative programs, and on the bad side, we have programs that are truly ugly or difficult to use.

Of course, the user interface does not stop at the screen, but includes the entire environment, from the organization of the files required for operation, to the setup required, and to how the program behaves when you exit.

On the good side, we do have beautiful and imaginative programs, and on the bad side, we have programs that are truly ugly or difficult to use...

Personally, I like programs that look for needed files in an 'assigned' directory, and preferably in a directory that is assigned by the Amiga on startup, like s:, l:, devs:, and so on. Many manufacturers seem to think that you will always boot from their floppy, and make some unwarranted assumptions on that premise. They will look for a file in a directory with a special name, meaning that you either have to boot from their disk, play games with the Assign command, or even use a file zapper to change their program. How much trouble is it to look first in the current directory then, if the file is not found, look in a standard system directory? I can tell you from experience that it adds one line of code to the source if it is written in C or Modula 2, and the benefits of doing it are obvious. It eliminates special setups and assigns, and thus relieves the burden on users, most of whom would rather just type a program name or click on an icon to get started. It also allows a lot of flexibility in where you put the main program. I like my terminal programs in the c: directory, while others like them in a directory reserved for terminals. A little thought on the part of the author will allow all users to set things up and run the way they want to.

While I am thinking about files, let me take the opportunity to chastise those who insist upon files having mandatory extensions. I will grant that files with extensions make things easier for the programmer and prevent you from inadvertently trying to calculate your mortgage payments from an IFF picture file, but that is almost irrelevant - the user is the one for whom things should be made easy, and most users will save files with a name that is easy for them to remember, with or without extensions. If you are developing a program and feel that you want filenames with extensions, please have the courtesy to allow the option of turning off the 'mandatory' switch.

Menu etiquette

Menus are often a weak point in programs. The COMMSEQ flag allows you to specify a Right-Amiga key sequence that performs the menu selection. It costs nothing in terms of code size or speed to implement this, but will be very much appreciated by a user who likes the keyboard shortcuts. Using standard names for menu items is another important factor. If I have to stop and think about what a menu item really means because it uses 'Archive' instead of 'Save', or 'Copy region as Kill' instead of 'Cut', it reduces my efficiency, is a potential cause of errors, and causes me to have less respect for the offending program.

Gadgets are offered in a variety of styles - Boolean, String, and Proportional - and thought should be given to what type is best suited to the job at hand. A programmer should not be afraid to use more than one gadget for the same job, possibly offering presets as well as string gadgets. A proportional gadget should, whenever possible, be 'real time', showing the changes as it is moved. Charlie Heath's file requester offers a good example of this. If up and down arrows are provided, they should repeat when appropriate.

The way a program exits is important. Does it come back to where you started it, to the same environment? I once used a program that didn't bother to restore my current directory to where it was when I started it. I didn't realize it at first, and only became aware of it when I ran it, and then tried to delete a file after it had exited. Luckily, there was no file of that name in the place it left me, but there could have been, and that was enough to make me think twice about whether I wanted to leave the program on my machine.

All programmers hoping to have their efforts sold, or at least appreciated, would do well to look at the programs that are well liked by users, and the process should not stop there. Just because a program is finally out the door and on the store shelves doesn't mean it's finished. Improvements, bug fixes, and user support will help keep a program selling, and generate user loyalty to a product and company. Listen to users' complaints and wish lists, implementing those that are good ideas. In short, do the best you can for the user. You will be doing the best for the Amiga community and your own company, and we will all profit from it.

b i t s

Got an interesting programming tip, a short routine, or an unknown bit of Amiga trivia? Send it in - if we use it in the bits column, we'll credit you in the column and send you a free one-year subscription to Transactor for the Amiga.

Easy Rule for Minterms

A few of the Graphics Library's functions that give you easy access to common blitter operations use a "Minterm" argument to specify the logical operation the blitter is to perform between the source and destination bitmap areas before copying the result to the destination. The functions are:

BltBitMap()	blit from one BitMap to another
BltBitMapRastPort()	blit from a BitMap to a RastPort
BltMaskBitMapRastPort()	blit from a BitMap to a RastPort through a mask
ClipBlit()	blit from one RastPort to another

Functions that blit to a RastPort do proper clipping of the image to within the RastPort.

With these functions, bits 4-7 in the Minterm describe what to do with each bit in the destination based on the current state of the bits in the corresponding position of both source and destination. The operation will take place for each bitplane in the BitMap(s), or for those bitplanes specified by a "Mask" parameter to the function or the Mask variable in the RastPort. The source and destination BitMap for these operations may be one and the same.

Perhaps the most common Minterm is for a straight copy from source to destination, a value of C0 in hexadecimal (usually expressed as 0x0C0 in the C function call).

Minterms for other possible operations are very easy to determine using this simple truth table:

		Destination	
		1	0
Source	1	m3	m2
	0	m1	m0

The squares designated by "m0" through "m3" represent bits in the Minterm and determine, for each bit being blitted, what will be put in the destination area when the bit currently in the source and destination is as indicated by the left column and top row in the table. For example, we could formulate the vanilla-copy Minterm by the following rules: when the source contains a zero, the result should be a zero; when the source contains a one, the result should be a one. To make this true, m3 through m0 should contain the following bit pattern:

m3	m2	m1	m0
1	1	0	0

This is the value put into bits 4 through 7 in the minterm, yielding the common vanilla-copy Minterm 0x0C0. As another example: a common operation is an exclusive-or, used to change a bitmap so that the same operation can restore it to its original state. An exclusive-or between the source and destination dictates that if a bit in either is set, the result should be a one, except in the case where they are both set. Filling in the truth table, we get:

m3	m2	m1	m0
0	1	1	0

...which yields a minterm of 0x060 for the exclusive-or operation. Other useful Minterms we can extrapolate are 0x000 for clearing all bits in the destination rectangle, and 0x0E0 for an OR between source and destination.

All sixteen combinations derived from the truth table are listed below, with those of greater use indicated by an asterisk.

Minterm bits	Operation Between Source and Destination
0000	* clear all bits in destination
0001	inverted OR
0010	set bits where source=0 and destination=1
0011	* copy inverted source to destination
0100	set bits where source=1 and destination=0
0101	* invert destination
0110	* exclusive-OR
0111	set all bits except where src and dest overlap

```
1000      set only bits where source and destination overlap
1001      inverse of exclusive-OR
1010      same as destination - has no effect
1011      set all bits except where source=1 and dest=0
1100    * vanilla copy from source to destination
1101      set all bits except where source=0 and dest=1
1110    * OR operation
1111    * set all bits in destination
```

From the above, we can put together some useful defines for a "minterms.h" header file:

```
#define MINTERM_CLEAR          0x000
#define MINTERM_INVERT_SRC     0x030
#define MINTERM_INVERT_DST     0x050
#define MINTERM_XOR            0x060
#define MINTERM_COPY           0x0C0
#define MINTERM_OR             0x0E0
#define MINTERM_FILL           0x0F0
```

To explore the actual results from the blitter operations with any given Minterm, the program "Minterm Tutor" is on the Transactor disk for this issue and is generally available. It lets you click on one of sixteen Minterm gadgets and see the results of the blit between single-bitplane source and destination images. The images are circles of different sizes and positions; the result is a Venn-diagram showing the effect of each Minterm graphically. The filled-in truth table and description of the Minterm's effect is also displayed when the blit is done.

Tiny-Type

Who says you need a lot of code to do anything on the Amiga? This program copies from its standard input to its standard output and can be used as a TYPE command - just use the redirection operator "<" before the filename. Redirect the output as well to copy to another file instead of printing to the screen. All the work is done with a single line of C code - this is the entire program:

```
/* Tiny Type - Manx users: use 32 bit integers */
#define BUFSIZE 512
char buf[BUFSIZE];
main ()
{
   while (Write(Output(), buf, Read(Input(), buf,
         BUFSIZE)) == BUFSIZE)
      ;
}
```

Race Conditions
Rico Mariani

Whenever you're using public message ports there's always the danger of race conditions. If you do a FindPort() and then a PutMsg(), you can't really be sure that the PutMsg() will work because the port might be deleted between the time you find it and the time you put the message. Similarly, if you're about to delete a port (with DeletePort()) you should be careful that the port is empty of messages before you delete it, but if you check the port first and then delete it, how do you know that a message didn't show up just after you checked? In both these cases you must surround the code with a Forbid()/Permit() pair. Forbid() will disable multitasking so you know nothing can happen until the call to Permit().

An Intui-Bug
Rico Mariani

Here's an exciting way to crash your system: while dragging a window on the Workbench screen, press Left-Amiga-N or M (this will work even if there isn't a second screen up). Continue dragging the window to the right as far as you can go... Whoa! Gets a bit carried away, doesn't it? If you were to let go of the window while holding it out beyond the edges of the universe here, truly nasty things would happen; do so only if you don't mind a re-boot in the very near future. If you drag the window back within the sane limits of the screen and let it go there, it will rest happily as usual and no harm will be done.

Double-Buffering Under Intuition

Animation on a multiple bitplane screen can often flicker and flash unacceptably because one bitplane is moved at a time, creating unwanted intermediate images. Double buffering - rendering to alternate display areas and only displaying an area when the rendering is completed - is the solution. Creating a double-buffered display using a standard slideable Intuition Screen doesn't seem to be documented too well, but it can be done in several ways.

You must have two BitMap structures for double buffering: the one that Intuition created for you when the Screen was opened, and another one of the same size and depth. You will be switching between these two BitMaps when double buffering, rendering your next frame to the BitMap not currently visible. Following is a code fragment to show the basics.

First, some important variables:

```
struct Screen *screen;      /* pointer to screen structure    */
struct BitMap *bm1, *bm2;   /* pointers to the two BitMaps    */
struct BitMap dbuf_bm;      /* used for double-buffering      */
```

Open the Screen as usual (assuming a properly filled-in NewScreen structure called newscreen):

```
screen = OpenScreen(&newscreen);
if (screen == NULL)
   myerror("Couldn't open Screen!"); /* your exit routine */
```

Next, store pointers to the BitMap structure used by the Screen, and to the alternate BitMap:

```
bm1 = &screen->BitMap;
bm2 = &dbuf_bm;
```

Now create a new BitMap the same size as the Screen's to use for double buffering:

```
/* width, height and depth for alternate BitMap
   (same as Screen) */
#define BMWIDTH     screen->Width
#define BMHEIGHT    screen->Height
#define BMDEPTH     screen->Depth

/* set members of BitMap struct using the InitBitMap() */
InitBitMap(bm2, BMDEPTH, BMWIDTH, BMHEIGHT);
/*allocate space for bitplanes and put ptrs in Planes[] array */

for (i = 0; i < BMDEPTH; i++)
    if ( (bm2->Planes[i] = AllocRaster(BMWIDTH,
        BMHEIGHT)) == NULL)
        myerror("Couldn't allocate memory for bitplanes!");
```

Okay, now the screen is up and we have two BitMaps: one displayed on the screen (bm1), and another in reserve not being displayed (bm2). Your animation routine should alternate between rendering in one, then the other (you may want to make a RastPort for the second BitMap to use the RastPort-based Graphics Library calls). After you complete rendering to a BitMap, it should be displayed. This is the part that's a bit tricky; the ROM Kernel Manual would have you switching around pointers to Copper Lists to do it, which drives Intuition bananas. The easiest way to do it is as follows:

First, you have to change the screen's idea of which BitMap structure is being used. This pointer is found in the RasInfo structure, which can be found through the Screen's ViewPort, which is contained in the Screen structure itself. In short, you can change the BitMap pointer like this:

```
screen->ViewPort.RasInfo->BitMap = bm2;
/* point to alternate bitmap */
```

That's not all there is to it - you have to actually get the system to change Copper lists to display the new BitMap. Here's the easy way:

```
MakeScreen(screen);  /* does a MakeVPort() */
RethinkDisplay();    /* let Intuition do ALL the work */
```

The call to RethinkDisplay() lets Intuition think all about the current relationships between the ViewPorts and other things, and do whatever it has to. Intuition will be happy using this method, and you can slide the screen up and down all you like during the animation without a flicker. The switching of the display will also be timed with the vertical raster beam so that a flicker is not caused by changing the display while the raster is in mid-screen.

The down side to this is that RethinkDisplay() is a pretty major function: the Intuition manual warns that it can take several milliseconds to run, being especially hard on multitasking by doing Forbid()/Permit() a few times. This seems to be the safest and cleanest way to do double buffering, but sometimes you can't spare the CPU time.

The second method is to use the Graphics Library calls directly. This uses less CPU time, but doesn't keep the animation's composure too well while the screen is sliding up or down. This isn't too important, because as soon as the Screen is still the animation will smooth out again.

To use this method we need a pointer to the current View. This can be done early on in the program, finding the pointer from Intuition:

```
struct View *view; /* global variable declaration */
view = ViewAddress();
```

After you switch the BitMap pointer in the RasInfo as above, do this:

```
MakeScreen(screen); /* new ViewPort Copper instructions */
MrgCop(view);    /* merge Copper instructions into View */
LoadView(view); /* put new View into effect */
WaitTOF(); /* wait for vertical retrace */
```

These are the methods that we've found to work through experimentation. The first method - calling RethinkDisplay() - seems perfectly safe, if time consuming. The latter method looks a bit scary while the window is sliding and may not be completely kosher. A third method is to call ScrollVPort() instead of MrgCop() and LoadView(), but that seems as slow as using RethinkDisplay(), and can be unpredictable. If anyone knows of a cleaner or faster way to do double buffering on an Intuition Screen, please let us know.

Reading Workbench Arguments

Making a program runnable from the Workbench is easy: just make an Icon for it. The next level of Workbench support for a program is to accept arguments from a "project" Icon that is double clicked to load the program, or from multiple projects that are shift-clicked along with the program's Icon. Reading the names of the files corresponding to these Icons is also simple, simpler than finding out how to do it from the ROM Kernel Manuals is.

Reading Workbench arguments is analogous to reading CLI arguments, except that each argument is an Icon instead of a character string. Within each Icon can be found a "TOOL TYPES" field where information for the program can be found, but the name and directory of the file that the Icon corresponds to is often all your program needs to know.

For example, a program that processes a number of files could read all its command line arguments and treat each one as a file name. Ideally, such a program would do the same if run from the Workbench, working with files that had Icons. The CLI-driven program might handle its arguments like this:

```
main (argc, argv)
int argc;
char **argv;
{
int i;
for (i = 1; i < argc; i++)
   do_something_with(argv[i]);
}
```

The parameter "argc" will contain one more than the number of arguments passed on the command line. argv[0] holds the name of the program itself, with argv[1] through argv[argc-1] holding the arguments. If the program was run from the Workbench, argc will be zero. Your program can test for this and use the Icons passed to the program as arguments. This is not much more difficult than reading the command line arguments. Here's the code - the explanation will follow.

```
extern struct WBStartup *WBenchMsg;
/* Workbench startup message */
main (argc, argv)
int argc;
char **argv;
{
int arg;
  if (argc == 0)
  /* called from Workbench - read Icons as args */
    for (arg = 1; arg < WBenchMsg->sm_NumArgs; arg++)
    {
      CurrentDir(WBenchMsg->sm_ArgList[arg].wa_Lock);
      /* set directory */
      do_something_with(
            WBenchMsg->sm_ArgList[arg].wa_Name);
    }
  else      /* called from CLI - read args as usual */
    for (arg = 1; arg < argc; arg++)
      do_something_with(argv[arg]);
}
```

The WBenchMsg is passed to a process when it starts through a message sent by Workbench, and is supplied by the C startup code. In this structure, among other things, can be found the number of arguments passed, and an array of "ArgList" structures that contain the name and directory Lock for each argument.

Note the call to CurrentDir() - this is so the file represented by the Icon can be opened regardless of which directory it's in.

The Workbench arguments are analogous to command line arguments in that the the first one (subscript zero) points to the name of the program itself, while subsequent ones point to the arguments. The difference, with the above code, is that the arguments can only represent the *names of files* whose Icons were selected by the user before the program was run. To read other arguments from the Workbench like flags, options, etc., you should read the "TOOL TYPES" field of the Icon, which is another topic. (See Rob Peck's article in this issue.)

To force a project Icon to load your program when double clicked, the whole pathname of the program should be put in the Icon's "DEFAULT TOOL" field (you can do this via the Workbench "Info" menu selection). If a program is run in this way, the project Icon will be passed to the program as the sole argument.

Disk Rescue
Bryce Nesbitt

If you ever get a disk that has a "Read/Write Error", *don't panic!...* there is an easy trick that can save the day in an amazing number of cases.

Simply reach *into* the front of the drive, and grab the edge of the disk. Pull it up or push it down a bit and "Retry" the operation. It may take several attempts and several different positions, but chances are you will get a successful read!

The reason this works is related to alignment. Many disk errors are caused by slightly different adjustments of the Read/Write heads in different drives. By flexing the disk in this manner you slightly change what part of the disk passes under your drive's Read/Write heads, compensating for the problem. Be sure to back up the flaky disk as soon as possible!

Single Drive Disk Copy
Graham Reed

Have you ever wanted to format or copy a disk when you have booted with a software package's Workbench, only to be told that Workbench "Can't open SYS:System/Format" (or DiskCopy)? Well, just because Format and DiskCopy tell you to select an Icon and use the Menus, doesn't mean you have to resort to the CLI and (ugh) typing. Besides, some Workbench disks don't even let you get a CLI.

So, if you have a one-drive system, just put a normal Workbench in your drive (i.e. the one that came with your system), and open its System Drawer. Now insert the disk you would like to copy or format, and select its Icon. Using the extended selection method, hold SHIFT down and double-click on Format or DiskCopy, depending on what you want to do. Follow the requesters, presto! A blank or duplicate disk!

For you (lucky) two-drivers out there, formatting is essentially the same as the process described above, but you don't need to swap disks. To DiskCopy, open the System Drawer as above, and insert the source and destination disks in your drives (it doesn't matter which ones). Now, the order of selection plays a very important role here, because it decides which disk is the source and which is the destination; write protect the source disk for extra security. Now, click once on the source disk Icon to select it, then hold down the shift key and click on the destination disk. Still holding SHIFT, double-click on the DiskCopy Icon. Follow the requesters as usual, and there you go!

Amiga Dispatches

Emulation, A2000 keyboards, promising rumours

by Don Curtis

Don Curtis is a police officer in Denver, Colorado, who has for the last two years been assigned to program design and development along with system design and maintenance of 10 AT&T Unix 3B2 computers. In his spare time, Don is an assistant sysop on CompuServe's AmigaForum.

Good News for Canadian SideCar owners...

If you have the West German version of SideCar attached to your A1000, here's some good news for you! You can use the new software from the A2088 (BridgeBoard) with your SideCar and get dramatic improvements. Some of the improvements you'll get by installing the new software are: No debugging messages are dumped out the serial port on startup of PCWindow (mono or colour), and these programs load and run faster. SideCar works with expansion memory. SideCar now runs even when the Amiga side is being heavily used (it had been reported in the past that heavy Amiga-side usage would hang the PC side up). File transfer from the PC to the Amiga and back again now works fine. There are no more limits on disk partitions on the PC side for the Amiga (apparently, it used to be that a PC disk would choke on the Amiga partition when it reached about 60 per cent full).

To realize these improvements you'll need to get a copy of the A2088 Install disk from your Amiga dealer. Now, there is a question as to whether the dealer can make you a disk as an upgrade, or if you need to purchase the new software. Check with your dealer; they can contact Commodore Canada for permission.

Once you get the A2088 disk, look for the script file called SideCarInstall. Make a copy of your 1.2 WorkBench disk, then execute the script file SideCarInstall on the A2088 disk. That should create a new SideCar boot disk with all the new software. Don't use the BridgeInstall script file, as that will copy the wrong LPT1 program; for the SideCar, you'll need the program LPT1S. If you've been using a partition on a PC Side Hard Disk, you'll have to reformat the partition. Make a backup of all the data on the present partition, reformat, then copy your files back. You should now be back in business with a whole new machine... or so it will seem, according to one user's report.

A bit more on the BridgeBoard...

A few folks have been having problems hooking up a speaker to the BridgeBoard. The output of the speaker jacks (J2) isn't enough to drive a normal speaker... at least, not with reasonable volume. I looked at it with an oscilloscope and, on my machine, I got pulsed output at about 2v peak to peak. That's plenty to drive an amplifier, but then you need to bring the speaker line outside the A2000's case. There are a few alternatives. You can run a line from the right pin of J2 (viewed from the component side) to pin 6 of the internal serial connector. The internal serial connector is on the motherboard, right behind the external serial port. This will give you PC audio output from the Amiga's normal audio jacks.

Another method would be to use a piezo buzzer. Radio Shack sells several styles of them, and I tried a few. The output was tolerable, but not great. I didn't get a chance to try all the types Radio Shack had to offer, but suspect there's not much difference between them. The method I ended up using was to dig up an old cassette recorder microphone from my junk drawer. Yup, a microphone - most of them work as fairly good speakers also, and this one was no exception. It was the best of all the alternatives I had tried and, being small, easily fit inside the A2000. If you do use a speaker/microphone, make sure your placement of it inside the A2000 doesn't interfere with the A2000's operation. I initially placed mine on the front of the motherboard within a couple of inches of the 5.25" drive. That was close enough that the magnetism interfered with the drive, and I got lots of errors. I moved it to the left of the BridgeBoard and everything works fine now.

Rumours about the BridgeBoard continue to fly. Word has it that CBM has stopped making the 8088 version (A2088) in favor of the 80286 version (AT style... A2286?). At the time I'm writing this, I've been unable to confirm that, but do know that CBM is working on an AT style BridgeBoard. The rumoured price for the AT style is in the neighborhood of $1,000. If that is true, it doesn't make sense to me... but doesn't surprise me, either - CBM isn't known for always making sense. While

there certainly are folks who want an AT style BridgeBoard, I wonder how many of those who want PC compatibility are willing to pay the extra $300 for the AT style board when all they really need is the original XT style BridgeBoard. I'd hate to see Commodore lose sales of the A2088 while awaiting release of the new BridgeBoard. Right now, demand for the A2088 remains well ahead of supply. I think there is room in the marketplace for both versions, just as there is room for both the A500 and A2000. It all depends on what the users want, and Commodore should be ready to supply either version

Emulators and clones...

Speaking of the BridgeBoard brings up the whole idea of emulators and clones. Are they a good idea? There's no hard and fast answer. On the one hand, PC compatibility is a good thing. Like it or not, MS-DOS PCs are here to stay and are a major force in the marketplace. They are the de facto standard for general business use. For the next few years anyway, if you bring work home from the office, you will more than likely need a PC at home to do that work. The Amiga and the Mac are slowly making their way into businesses, but at this point the PC has no real competition. With that in mind, adding PC compatibility to the Amiga is good. It means that folks who want an Amiga but need to run MS-DOS can have both in one machine. The BridgeBoard, being a PC on a board for the Amiga, is a viable and cost-effective alternative to a separate PC in the home.

What about the C64 emulators? Would an Atari emulator (ST or 8 bit) or an Apple II series emulator be a worthwhile idea?

But what about the other emulators? If Magic Sac for the Amiga ever gets here, is Mac compatibility a good thing? What about the C64 emulators? Would an Atari emulator (ST or 8 bit) or an Apple II series emulator be a worthwhile idea?

I think not.

Take a look at Transformer, the original PC emulator. It was slow, only partly compatible and, all in all, not very useful. If you showed someone your Amiga running PC software under Transformer, they'd snicker and say something like, "Gee... that's cute, but I can't wait 20 minutes for that spreadsheet to do a simple recalc that only takes me 5 minutes at work!" The C64 emulators are in the same boat. They work, sort of, but slowly. They aren't completely compatible and probably never will be. We don't have Atari 8 bit or ST emulators, nor do we have Apple II or Mac emulators, but if we did, and the emulation was in software, they'd be as slow as or slower than the other emulators we do have.

Anyone seeing the Amiga running one of these slow emulators could possibly come away with the idea that the Amiga wasn't even as good as the machine being emulated, and we all know that's not true. They might even get the idea that there is no good software for the Amiga itself, so folks have to resort to emulators to get the software they want.

What about hardware emulators? They should give you the same speed and compatibility as the BridgeBoard gives us for PC emulation. For the 8 bit computers, like the Atari 800s and C64s, or even the Apple IIs, the cost of a hardware emulator would more than likely be close to or higher than the real thing. It just wouldn't be cost effective.

That leaves us with hardware emulators for the Mac and ST. Would we gain anything? Is there a real need to emulate either machine? Again, I don't think so. Magic Sac for the ST is being sold to those who either want a cheap Mac or want to run software that can't be found for the ST itself. Do we really want folks buying an Amiga so they can turn it into a cheap Mac clone? As far as the ST is concerned, the Amiga A500 is already lower in price than the ST, and software that is available for the ST is also available for the Amiga. Nothing would be gained.

Finally, what if you already have one of the other machines and want to move up to an Amiga, yet keep your old software? That's probably a valid concern. My experience, though, is that folks who move up to a new machine soon tire of their old software and buy new software anyway. In the long run, they will probably find that it's cheaper to keep the old computer. The price they'd get from a trade-in or outright sale wouldn't be much different than the cost of an emulator for the Amiga. When they tire of the old computer, they'll have both the computer and software to go with it... a definite plus when trying to sell a used computer. Selling a computer at one point, and trying to sell off the software later, just doesn't work out very well. As an example, I just sold off the ST I had been using for software development. The first thing folks asked was how much software I had to go along with the computer. As soon as they found I only had development tools (compiler, text editor, and so on), they quickly lost interest or asked for a lower price.

I realize that these same arguments against emulators can be applied to the BridgeBoard as well. It's only because the PC is so entrenched in the business world (and in many people's minds) as 'the' computer, that the BridgeBoard, among all the possible emulators, makes any sense. There are just too many PCs and too much PC software out there to be ignored.

And speaking of clones...

Interesting item. Discovery International sells Marauder II, a disk copy program that will clone (copy) almost any program on the market. Discovery even includes routines that unprotect certain programs so you can install them on a hard disk. There are uses for utilities like this and, when they are legitimately

used, I have no problem with them. The manual for Marauder II starts out by stating that the Federal Copyright Laws allow the owner of a program to make an *archival* copy. It goes on to state that, with certain programs, you only obtain a licence to use the program, thus are not permitted to make an archival copy. Okay, so far, so good. Now comes the rub: guess who recently published a program that can't be copied by Maurader II? Yup, Discovery Software recently published Arkanoids, and Marauder II won't copy it. There is no licence clause stating that you don't own Arkanoids, nor is there any statement that you are not permitted to make archival copies of the disk. I'm not knocking the software - either program - they're both good. I am just curious as to why a company would publish a program that will duplicate almost anyone else's copy protected programs, but not their own.

I will say this in defence of Discovery: they do include a coupon with the program that allows you to get a backup copy for three dollars. That's a very reasonable price and policy, and it does allow you to have a second copy of Arkanoids in case your original fails.

A2000 keyboard fix...

A few of the new Amiga A2000s are coming through with a minor but annoying keyboard problem. It seems that the first (and only the first) keystroke in a new CLI window gets lost. Luckily, CBM has provided an official fix. First, it *only* concerns the Hitek keyboard. The Hitek can be identified in several ways: the function keys are larger than the keycaps on the typewriter portion of the keyboard and, if you look closely between the spaces between keycaps, the underlying shield is black. Of course, the final way to determine which keyboard you have is to see if you lose the first keystroke in a new CLI window.

Note that doing this mod yourself will void your warranty. If your A2000 is still under warranty, return it to your dealer's for the fix. Dealers have been authorized to make the modification as a warranty repair.

Even if your A2000 is out of warranty, you should probably contact your dealer to make the modification - it's safer in the long run than doing it yourself. That way, if anything goes wrong, the dealer will be responsible for any repairs that may be needed. The modification requires the removal of the drive bay/power supply bracket and removing 2 capacitors (c910 and c911) from the Motherboard by the keyboard connector. This is *not* for those who are unfamiliar with electronics equipment repair.

MotherBoard revisions for the A2000...

I've talked to a few people who are concerned that they may have a defective or obsolete motherboard in their A2000 (the West Chester, or B2000, version of the motherboard has been revised a few times since it hit the dealers' shelves). All the revisions are minor changes and have no real effect on the operation of the computer. According to Dave Haynie of Commodore-Amiga (the designer of the B2000), no machines went out with rev 4.0 motherboards. A few machines, mainly dealer demo units, went out with rev 4.1 motherboards. These put out poor video due to overly large ferrite beads on the video output circuit. Very few of these machines went out to customers. If you had one of these A2000s with poor video, the fix is very easy: just remove the ferrite beads in the video output circuitry. The rev 4.2 motherboard corrected this problem. The rev 4.3 motherboard added 2 pull-up resistors for the Commodore (MOS) manufactured GARY chip. This modification was added while the Toshiba manufactured GARY chips were still being used so that no surprises would occur when the MOS parts started to be used. What's a GARY chip? It stands for GAte aRraY and is used in conjunction with Agnus.

The machine is also rumoured to support a 1024x920 non-interlaced display and have a palette of 2 million colours...

As you can see, the only noticeably different version of the motherboard was the original rev 4.1. If you had one of those machines, you would notice very fuzzy video. Looking on the rear of the motherboard, by the RGB connector, you would see 3 ferrite beads with wires wrapped around and through them. Rev 4.2 and 4.3 motherboards have the RGB output wires simply running through the beads, not wrapped around them. If you happen to still have one of these machines, take it to your dealer and they should remove the beads. There should be little or no charge for this.

What's new at CBM...

There are lots and lots of rumours about a new Amiga. I want to stress that, as of this writing, they are rumours only, not official announcements. The rumoured machine (A3000?) would be 68020 or 68030 based depending on the cost of the CPU when production starts. If it's a 68020 based machine, it will have a 68851 MMU; on the 68030, the MMU is built into the chip itself. In either case, the math coprocessor is rumoured to be the new 68882 chip. The motherboard will have 2 and possibly 4 megabytes of RAM as standard, and there will be more expansion slots. The machine is also rumoured to support a 1024x920 non-interlaced display and have a palette of 2 million colours. With a display resolution like that, each bitplane is going to use about a megabyte of RAM, so this may well be a very hungry machine as far as memory is concerned.

Commodore has also decided to support up to 4 megabytes of RAM on its A2620 coprocessor board for the A2000. The A2620 is the forthcoming 68020/68851/68881 board that will go into the A2000's coprocessor slot. It will come with 2 Meg of RAM standard, but be upgradable on-board to 4 Meg. The A2620 *has* been tested in an A2000 with the BridgeBoard installed, and both work just fine.

ACCESS

The world of non-commercial software

by Steve Ahlstrom

As the primary Sysop on CompuServe's AmigaForum, and a veteran Amiga programmer, Steve Ahlstrom sees just about every notable public domain and shareware program that gets written, and in this column he tells you about the ones that have recently caught his eye. If you have some information you'd like to share through this column, you can leave a message to Steve on AmigaForum (his PPN is 76703,2006), or - if you're not a CIS subscriber - send the info to us and we'll pass it on!

The Amiga is one of three computers shipped with a visual interface that is an integral part of the user friendliness of the machine. We are all familiar with it: point, click - and boom, something happens. While this is a tremendous advantage for novice users to get up and running as quickly as possible, and the Workbench is powerful in many areas, it does have some serious drawbacks. One of those drawbacks is the inability, from the Workbench environment, to be able to see, either graphically or textually, all of the files residing on any given disk. In my opinion, this is an almost unforgivable oversight.

Clouds usually do have their silver linings, however. Because of the weaknesses in Workbench, Amiga hackers the world over have turned their attention to writing programs that give you a visual interface into the Amiga file system. So much so that it appears at times that to be taken seriously as an Amiga programmer you must have written your own version of the almost mystical 'Directory Utility'. There are dozens of them available in the public domain and shareware world. I'm going to take a look at one of the best of them. I haven't yet found one that entirely meets my needs and idiosyncrasies, but I do use a couple of them regularly, and I'm sure that, given the vast number of variations on the theme, I will eventually find the one that is perfect for me.

THE CLI WIZARD
Written By J.L. White
© 1987 Merlin's Software

The CLI Wizard has tons of features, and most of the commands available are user configurable. Here are some of my impressions; remember, though, that this is subjective. What I like in a directory utility may be completely different from what you like.

There is a single window display of the directory that is currently the 'source' directory. It can display up to 13 file/directory names. There is a gadget whose image is that of a toggle switch. You can toggle this to make another directory the 'source' directory. You can then mark files to be copied between directories. Whichever directory is displayed in the window is the source, and the unseen directory is the default destination. I would have preferred to see both directories at once (two separate windows), especially since the displayed directory is the source directory. This can be confusing because next to the string gadgets where you define what directories to use there is an 'S' and a 'D' (for source and destination) before the strings. By toggling the gadget you in effect change which is which. If you're not careful or paying close attention, you could easily find yourself manipulating the wrong directory.

The file window has a scroll bar. Unfortunately the scroll bar is not interactive, meaning that the file/directory names do not scroll as you are moving the scroll bar. The list is updated only on release of the gadget. This would be acceptable if the author had also included up and down single step gadgets. With the current functioning of the scroll bar and the lack of single step gadgets, it can be difficult to locate a particular name in a large directory - it's easy to overshoot that part of the displayed directory that you want to see. In my testing I used my Sonix/Instruments directory with 182 entries.

I have no idea what the author is doing when saving the directory in memory, but on a large directory, the display can be slow. In my test directory of 182 files, with the directory cached in RAM (FACC II - thanks Perry!), it takes a full 10 seconds to read, sort, and display. My hunch is that the sort algorithm used is the bottleneck here, but 10 seconds with no disk access to slow things down is poor performance. The documentation states that directories with up to 400 entries can be read. While 400 entries, on the surface, may seem like more than you will ever need in any given directory, I don't feel comfortable with programs that put arbitrary limits on the user. Using a hard disk for storage, there may well be applications that would require 400 or more entries in a given directory. A better approach would be *no* memory limitations, not an arbi-

trarily derived one. By using a linked list of entries, memory could be allocated for each entry on the fly. If memory is not available an appropriate error message could be issued. For me, this would be a much more friendly approach.

When any operation is done, whether destructive (like a delete) or non-destructive (like a copy) the directory window is redisplayed from the first alphabetical entry. This is time consuming and not always convenient. In my view, a program that aims at user friendliness ought not do you this kind of arbitrary 'favour'. I would have much preferred the list to be redrawn only as much as necessary, keeping the current top displayed entry the same.

The CLI Wizard has 24 built-in functions and 24 user defined functions. Most of the built-in functions are self-explanatory, although some of them require that you have specific files in your c: directory.

Built-in functions

• ALL Selects all files in directory (including those not seen in the window).

• COPY Copies selected files from source to destination.

• MOVES Moves files from one directory to another on the same device. I have a bit of a problem with this one, which is really nothing more than a rename (hence, the restriction to the 'same device'). A true move should be a copy of a file by whichever means is most efficient. If it's to the same device, a rename is sufficient, otherwise the file should be copied and the original afterwards deleted. That's the real difference between a move and a copy. A copy places a duplicate of the file somewhere. A move transfers the file to a different location, eliminating the original.

• SHOW Displays IFF pictures, using any IFF reader named 'show' in the C directory.

• CLEAR Deselects all files in the directory.

• DELETE Deletes selected files from the source device.

• RENAME This is normally an intuitive function, but not in this particular program. There is a string gadget labelled 'E' in the lower left hand corner of the screen. Here is where you type in the name you want the file to be renamed to. Then you select the file you want to rename and click on the 'rename' gadget. It would be much more intuitive if the program would pop up a string gadget for you when you select 'rename'.

• PLAY This uses Mark Riley's Play command (a Sonix SMUS score player), which should be in your c: directory. Click on a score and a window opens. Now supply the instrument directory name, and it plays the song. You can also play script files. Click on the file and enter 'P=' in the directory name gadget.

• BYTE Shows the number of bytes in all selected files.

• SINFO Shows how much disk space is left on the source device.

• DELDIR Deletes a selected subdirectory. An error will occur if you try to delete a directory that is not empty. Enter the name of the directory to delete in the 'E' box, then click the gadget.

• READ Read ASCII files (doc files, source code, and so on). Click on files to read then click the gadget. You can use your own read command by editing the data file. If no command is found it will use it own.

• ROOT Reads the current root directory.

• DINFO Shows how much disk space is left on destination device.

• MAKEDIR Creates a directory with the name taken from the 'E' box.

• EDIT Edit ASCII files. This function loads a file called 'ed' (which may be ED itself, TXED or another text editor). Click on the file you wish to edit then click this gadget.

• PARENT Reads the current parent directory.

• PAK Add files to a pak file. Use the 'E' box for the name of the pak, then click on the files you wish to add. A custom window will display the output. (PAK is an ARC-like utility written by Mark Riley. It has, admittedly, some advantages over ARC but, in my opinion, also suffers from some *major* disadvantages.)

• ASSIGN Displays all assigned devices in a custom window. If you wish to reassign some device, just enter it in the 'E' box (for example, "C: RAM:C").

• PRINT Sends a text file to the printer.

• RUN Runs another program. Any output from the program will be directed to the CLI window you ran Wizard from. Select a file then click this gadget. This function will also accept an extension from the 'E' box.

• ARC Add files to an arc file. Use the 'E' box for the name of the arc, then click on the files you wish to add. A custom window will display the output.

• LISTARC List the files in an arc file. A custom window will display the output.

• UNARC Extract files from the selected arc file. A custom window will display the output. If there is an existing file with the same name, it will be overwritten, so Listarc to check before unarcing.

There are 24 custom gadgets that you can define yourself. These are displayed in two groups of twelve (because of screen space limitations); you use a 'flip' gadget to toggle between the two sets of custom function gadgets. The gadget names can be up to seven characters long. When one of these gadgets is clicked, it runs the program listed in its data field. The program name can include a device specifier, with a total device/path/filename length of up to 30 characters. If the file requires parameters, type the parameters in the 'E' box before activating the custom gadget. Programs that want a return (like Format) will *not* wait for the return - so beware.

To edit the custom gadgets, bring up your favourite text editor and load in the "Wizard.data" file.

The CLI Wizard has sound, some of it irritating, some of it useful. The most annoying sound is the digitized gunshot when exiting the program. This is cute the first couple of times, but gets old quickly. Luckily, you can turn the sound off from the program or from the data file.

There is a lot going on in this program and, for the most part, it's well thought out. Visually, it's pleasing. As you can probably tell, there are many things about the program I don't like or wish had been handled differently from both practical and aesthetic reasons. It is, however, a very useful product; I'd highly recommend its use while you're waiting for that 'perfect' directory utility program to appear. You may be waiting a long time; meanwhile, The CLI Wizard will handle your needs.

The CLI Wizard is shareware; the requested fee is $15.00.

File PAKer 1.0
© 1987 by Mark Riley

PAK is a relative newcomer on the Amiga scene. It's a utility for combining and compressing several smaller files into one large file. As such, the author is hoping that it will become the successor to ARChive. It has some features that are nice. I don't think, however, that it will surpass ARChive as the most popular compressor/librarian.

PAK's biggest strength is that a .PAK file decompresses itself with no need for an external program. The name of the distributed program as you'll find it on BBSs and commercial services is PAK.PAK. At the CLI prompt you just type "PAK.PAK". In a few seconds the files contained within the .PAK file will be extracted and ready to run. Here its advantage over ARChive ends. This is also, in itself, one of its biggest faults (more on the down side of auto-extraction in a bit).

.PAK files are not efficiently compressed. While any compression is better than none at all, .PAK files are significantly larger, sometimes by as much as 50 per cent, than the same files compressed with ARChive. As distributed, PAK.PAK contains 2 files: PAK and PAK.DOC. The PAK.PAK file is 7168 bytes long. The same 2 files, compressed and libraried with ARChive is 5781 bytes long. Depending on several factors the difference can be greater (much greater) or less. I've yet to see any .PAK file smaller than its equivalent ARChive file. For the vast majority of ARChive or PAK users, the use of these utilities is to combine and compress one or more files into a single file to shorten upload/download times. PAK is a loser here. Less efficient compression means longer download times and, if you download from a commercial service, that means a higher cost to you.

With ARChive, you are required to have a 'master' program, ARCHIVE (usually shortened to ARC) to extract the .ARC file. Besides extraction, ARC also lets you do various things to the .ARC files, such as test the integrity of a .ARC file, or extract just one specific file from the .ARC. There is a master PAK program (called PAK) that has many of the same features. The difference is that the master program (PAK) is *not* required to extract the files.

Okay, getting confused yet? I am! I'll try to explain the real problems and potential dangers of PAK.

Argument for PAK: You no longer need an external 'master' program to extract the compacted files. If you *do* have the master program, you have many of the same features and functions that you get with the ARC program.

Argument against PAK: Since the most widely used application for compressing file is for modem to modem file transfers, ARC is much more efficient. You get more compression, less time spent online, less money spent with Ma Bell or your favourite commercial service.

ARC is also compatible across machines. Should you want or need to download a file originally written on another machine, and that file is in .ARC format, you can download it, unARC it, and do whatever you like with it. Many times you'll have a need for text files or C source files written on another machine and ARC is a simple way to get them easily. For me, that's one of ARC's biggest advantages; PAK, an Amiga-only program, doesn't offer this.

The biggest 'gotcha' is potentially very dangerous. Without the master PAK program, you run the .PAK file without having any idea of what it's going to do. You can't even be sure that it *is* a compressed/libraried file. Let's say you've downloaded something called SURPRISE.PAK. When you type SURPRISE.PAK at the CLI, your surprise may well be that it starts formatting all of your disk drives! Do you really want to run an unknown program blindly?

To sum up, then, if efficient compression is important to you, if you don't want to blindly run an unknown file, or if cross-computer compatibility is important to you, PAK is not for you. It's a good idea, but problematic in this incarnation.

PAK is a shareware; the requested fee is $10.00.

BreakPoint Part Two

Second in a series on debugging

by Victor A. Wagner

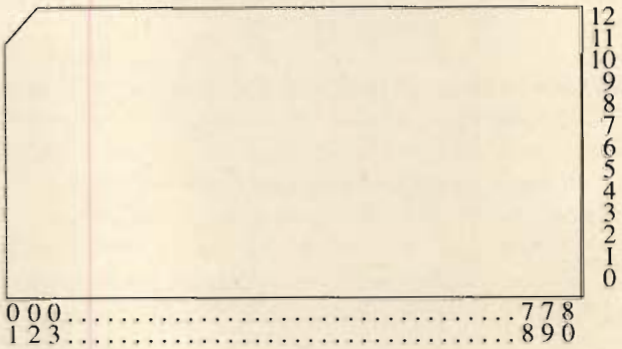
Vic Wagner started his association with computers in 1965 while in the US Air Force working in a group studying digital flight simulation. Since becoming a civilian in '66, he has been working mostly with minicomputer manufacturers in the real-time systems software area. Weekdays from 8 to 5 find Vic answering the technical support line for Computer Automation Inc., and maintaining three real-time software systems. Evenings and weekends he spends most of his waking hours talking to Taarna (his Amiga 1000), writing programs, and spending time on CIS AmigaForum (76046,3004). Vic is a recognized authority on Metadigm Inc.'s well-known MetaScope debugging program, and was recently elected President of the Amiga Friends users' group.

In part one of this article we examined early computer history, up to the late 1950s or early 1960s (Actually FORTRAN was introduced in the middle '50s). We learned that Testing was mostly ignored (as it still often is).

My own association with computers started in 1963, so much of what I've already discussed is hearsay; from here on it will be personal experience. Looking back at that time, it seems that not much has happened since I started in computers - compared to the early days, it seems quite mild.

I'm sure you're all *dying* to know the answer to the question I left you with at the end of part one. So far, every one of my friends who has read the article has looked at me at the end and said 'PRINT' or 'printf' or 'WRITE' - whatever the formatted output statement of their first HLL happens to be. I guess the question was too easy... That is, indeed, the answer.

Before the arrival of compilers (or interpreters in the case of BASIC), it was immensely difficult to generate output from a program. Actually it wasn't all *that* hard, but rarely would it take only one source line. Remember that we were still using things like paper-tape and cards for source. Those with cards were attracted to the **write** statement like bees to honey. The idea of inserting a single card into the program just to identify "I'm *here*" was valuable. I knew programmers who had a bunch of cards already punched that they used for debugging. These cards were a different colour than the normal ones used at their shop, and generally had the other corner cut off. What? you don't know what I mean about corners cut off cards? Well, for those of you who don't know what a computer punched card looks like...



...approximately, anyhow. The cards were 80 columns wide and 12 rows high. (I don't know why they were 80 columns wide, but that's why we *still* have things that use 80 character lines, like CRTs and line-printers.) The height of 12 rows is perhaps even harder to explain. Perhaps it will suffice to say that Hollerith designed it that way. In case you're wondering why the corner is cut off, it allows you to tell at a glance that the cards in a deck are aligned properly. If one of them gets turned upside-down or backwards, there will be a corner sticking out. If the debugging card is orange or red and the rest of the deck is 'natural', it will be easy to find and remove.

As a historical aside, here is the reason that punched cards are made to this particular size: Hollerith had convinced the Bureau of the Census to build a machine to help with the census. Late one evening, the workman (a machinist, I believe) came into Hollerith's office and said he could build the machine except for one tiny detail. The drawings had no dimensions for the cards that were to be used, so the drawers for them couldn't be made until that detail was supplied. Hollerith thought about this for a moment or two, then reached into his wallet and removed a dollar, handed it to the workman and said, "Make it this size". So the standard size for the computer punch card is the size of the early 20th century US dollar.

Anyway, the idea is to get at least *some* output from the program. When I'm writing and testing a program, the most wor-

risome time is until the program produces some output. If the program just goes into memory, flashes a light or two, then halts the processor (or aborts off the operating system), very little information is gained. We carefully examine our listing(s). We invariably decide that there is *no* way that our program cannot produce *some* output. So... we blame the hardware. It's malfunctioning. Or, we blame the compiler. It's generating bad code. Finally, in desperation, we add a couple of write statements to the program, saying something intelligent like "We're here" or some such. If we are very bright, it might even say something like "We're here 1", "We're here 2", etc. Since most programmers are a bit egotistical (I'm an exception), we tend to only put in a *few* write statements. Then it's back to the computer; compile, link, execute! Voila!

Huh?? *No output!?*

Ok, it's getting serious now. Let's take a close look at techniques for using this tool. I know, we all expect every program we write to work the first time. I guess programmers are all optimists. It seems we never put the PRINT statements in till after the program fails. I'm going to use 'PRINT' (the BASIC formatted write) to be representative of all the single-line formatted write statements. This is based on the assumption that all readers will be at least familiar with BASIC. There are some obvious first places to consider placing PRINT statements in the program.

1. Preceding a call to a subroutine. This one may be as simple as: PRINT "calling xyzzy"

2. Following a call to a subroutine. This one may be as simple as: PRINT "back from xyzzy"

3. Preceding an IF statement (perhaps of the form: IF a < b). For example: PRINT "compare <",a,b.

Note that in this PRINT statement we also print the values of the two things being compared. Since we can see which is smaller, we can tell what the computer will do when it gets to the IF statement. One small word of warning here about printing values: some languages allow *macros* or *functions* to have side effects. This means that the PRINT statement itself will *not* print the same values that the IF statement will see. This will generally cause you a lot of grief trying to determine why the program doesn't work correctly. It can also cause a program to miss some of its input. For example:

```
PRINT "comparing for equal",getc(),"="
IF ((c=getc()) == '=')
```

is going to read a character, then the IF statement will read another. The program will behave differently with than without the PRINT statement. This is probably not what you intended.

4. Preceding a CASE (or SWITCH) statement. Again, printing the value of the expression is useful. Be sure to read the warning in 3 above.

5. If your program is blowing up in a FOR loop, it is often valuable to include a PRINT statement that prints the loop index. For example:

```
PRINT "loop 1",i
```

or perhaps you can just add the loop index to some other PRINT statement you already have. A couple of things to watch out for here:

- putting PRINT statements in loops can cause a lot of output.

- when you examine the output be sure to check the first and last index values. This is where you will find those stupid boundary condition errors. You know the ones I mean: "Oops, that should be 0 to n-1, not 0 to n."

Another historical aside: Some of you have wondered why it is that so many examples of program fragments use variables with such names as: i, j, k, l, m, and n for index variables, or for limit checks and so on. The answer lies in the FORTRAN language (which was the first HLL available). In FORTRAN (well, early FORTRAN anyway) there were two different kinds of data - integer and real. There was no explicit variable typing as in modern HLLs (even later FORTRANs allowed some variations). So some wise soul decided that the type of variable would be determined by the first letter of the variable name. The letters that were picked were: I, J, K, L, M, and N (not surprised, were you?). Why were these letters picked, right out of the middle of the alphabet? Well, actually it's the letters from I through N, and it will be easy to remember the I and N because they are the first two letters of 'INteger'.

6. If you're pretty sure that the loop has no flaws, then:

PRINT "finished loop n" is probably in order.

The above ideas are for the initial run of the program. You probably don't want to include them all. What we are doing with this technique is making a small, problem-specific, trace program. It is easy to go overboard with the PRINT statements. You can print an entire box of paper without much effort at all (see part one for my comments on reams of trace output). This technique was fairly popular for several years. Remember that in the early days, the computer time was very expensive. Programmers were to submit jobs, then examine output at their desks. There were some installations where the programmers weren't even allowed in the same room with the computer. You handed your deck to some person through a small window along with some directions on how to run the job. Some time later you could pick up your output along with your source deck at another window. Or, if it was a very large installation, you would go find the pigeon-hole/table/bookcase assigned to your department and pick up the output there.

Sometimes turnaround time was measured in days. What did you do while waiting for your latest run to finish? You worked on another program. Maybe you worked on generating better

test data to run against your program. Then there was always the 'dreaded document' that needed writing.

At other installations, machine time was broken into some conveniently sized chunks (convenenient for the administrators, that is) and was apportioned out to the programmers by some algorithm. This algorithm was always so complex that a program could not be written to implement it. You submitted a request for one hour of machine time. The administrative staff took your request along with everyone else's, did some magic with them, then posted the Schedule. You would push your way through the crowd of programmers to read the just-posted schedule (lots of mumbling/grumbling going on here) to discover that you have been allotted one whole hour tomorrow morning from 0315 to 0415.

3:15 to 4:15 in the *morning!?*

Why did all the debugging schedules look like that? I suppose it was because the *important* jobs got run during the day. Like the 'computer usage report' and the latest incantation of the 'computer scheduling program' that some clown on the administrative staff was trying to write. Anyway, there it was, an hour of computer time... all to yourself. How did you know to ask for an hour? Most problems are generally found in about 10 minutes, the rest take several hours to find. The administrative staff has compiled lots of statistics on usage time and discovered that the average usage is one hour, so that's how time is allocated. You examine the schedule a little more closely. Who has the time just before you? How good are they? Are they likely to finish early? Maybe you should come in a little early and look over their shoulder? This last ploy was actually a very good idea. Either you would spot something (obvious to you because you haven't been staring at it for weeks on end) or the other person would be so embarrassed at not finding the problem that they would decide to quit.

"This problem's really got my head spinning. I think I'll give it a rest", they would declare. They would gather up all their stuff, thank you for your assistance, and wander off.

Perhaps the person before you was an attractive member of the other sex and/or just fun to be with... Anyway, whatever the reason, you show up early. When you arrive, you glance into the computer room, decide on which strategy you're going to use tonight (to get the machine early, visit, or whatever), grab all your appropriate materials (source deck, test data, and so on) and head into the computer room. There are some formal sounds, perhaps an obligatory magic incantation like:

"Couldn't sleep, so I came in early. Need any help?? <yawn>, or perhaps I could get you some coffee <another bigger yawn>?"

If this computer room actually allows people to drink coffee, you stagger off in the direction of the coffee source to get it without waiting for an answer. If you're very, *very* smart, you already know the other person's coffee preferences (you've researched it long before now). When you return, you sit down, slurp a bit of the coffee and in your best down home friendly voice say, "How's it goin'?"

I haven't met a programmer yet who won't be willing, yea, even eager to tell me what problems they are having with this intransigent program they're debugging. Beware at this point. The other programmer will try to intrerest you in *his* problem; make it sound like the most interesting problem you've ever seen. Now there is nothing wrong with learning about someone else's program. You might even learn some coding or debugging techniques. Listen carefully to the arguments given for this being the most fabulous problem ever worked on (you may need them yourself, someday). Appear to be taken in, actually you might be able to help. Keep one eye discreetly on the clock, however; you already know how time flies when you're involved with a computer.

As *your* scheduled time approaches, note carefully the comments and suggestions of the other person. They're certain to try to cajole you into allowing them to use some of *your* time to find this 'one last bug'. This moment can be used to learn additional techniques for wheedling time from the person scheduled after you. So, even if you don't find the other person's problem, there are opportunities to learn here that shouldn't be missed.

We will suppose that you are finally at the machine, ready to debug. First off, you will compile/assemble/link your program if you've updated it. Okay, now you have an executable test program, the appropriate listings, link map, and reference book/card on the debugging program you will be using. All set to go.

Most (if not all) computers can be made to execute a program at full speed then stop at a predefined location and return control to a program called a debugger. In many cases, this is accomplished by altering the program in memory and replacing the instruction at the desired location with a special instruction. In other cases the hardware has the capability to alert another program if a specific location is accessed. Whatever mechanism is used, some bright programmer has already found out how to do this and wrapped a program called a debugger around this feature, which is called a *Breakpoint*. These programs all allow you to specify a location to start executing at and at least one location at which to stop. Many (if not all) will allow you to just specify new stopping locations and will use the last location stopped as the place to start.

There is a fundamental difference in how to attack the debugging depending on whether you have an assembly program (perhaps your HLL compiler produces an assembly-like listing) or whether you have only the HLL listing. Let's examine the assembly case first.

If you have a complete assembly listing then you can 'see' what should be in each location of the program you are debugging. You may have to refer to the link map and do a little

arithmetic (perhaps your debugger will do it for you) to determine *exactly* where each instruction is, but that's not an insurmountable problem. The way I tend to debug in such a case is as follows: I read the listing from the beginning of the program, simulating the actions of the computer until I come to a conditional instruction of some kind. It is here that I place my breakpoint. I may not have a clue as to what the results will be when I get to the breakpoint, but I am *sure* that we will get there. I then start the program. Almost before I can get my fingers off the keyboard, the breakpoint has been reached. At this point I examine the computer registers.

Most debuggers today will automatically display the registers whenever a breakpoint is reached. In the earlier days, however, output was almost always to a slow typewriter-class device, so the registers might not be displayed automatically. At any rate, examine whatever is about to be tested to determine which way the conditional instruciton is going to behave. If the test looks like it is examining the right thing, but the value doesn't make sense, you should find out why the wrong value is being tested. Most debuggers will allow you to modify memory, so you can 'patch' your program to execute correctly the next time through this code. If (as is likely) there is some uninitialized variable, you can alter it now, perhaps by altering the register that contains it, before continuing to debug.

Breakpoint debugging is a succession of placing breakpoints at locations you are reasonably sure the computer will get to, examining variables and registers, and making patches to the program. If you're clever at this point, you will also mark up the listing with whatever patches you have made, so you can correct the source later. When a subroutine is called, you have the choice to breakpoint through the subroutine, or just to 'break' at the location where it will return. The choice depends on how sure you are that the subroutine has no problems.

A word of caution about breaking *at* subroutine entrypoints here. Most modern machines call subroutines without altering the memory at the subroutine entry point, so breaking there is no different than any other breakpointing. Some earlier machines (including some that are still being built today) have a subroutine call that places the return address at the subroutine entry point and starts executing at the location following the entry. This instruction is generally called JST or STj (Jump and STore, STore and Jump) or some such. If you are debugging on such a machine, *be sure never to breakpoint exactly at the subroutine entry.* If you do, your breakpoint instruction will be destroyed when the return address is placed there, and you'll never execute it anyway.

If you have only a HLL listing, another technique is required. I make extensive use of the link map. This document should list all of the globally-available subroutine entry points. You will also need, from your compiler manufacturer, some information on how data is allocated and how subroutines are called. You will need to know where the arguments to subroutines will be, where the results of functions will be returned, and how local storage is allocated. Armed with this information, you can debug quite successfully using this technique. Instead of breaking at every conditional instruction, I break *only* at subroutine entry locations (and at the returns... more about that later). From the listing you should be able to determine a subroutine the program must call no matter what. Perhaps there is a list of several subroutines, one of which will be called. No matter, just put breakpoints at the beginning of all of them.

When you get to the start of the subroutine, examine the arguments that have been passed. If they are correct, you can find the return address (you should know where it is kept) or you can find a subroutine that this one calls. At any rate the location of your next breakpoint shouldn't be too difficult to establish. Of course, if the arguments to the subroutine are wrong, the code between the previous breakpoint and the call should be carefully examined. Here is where it is handy to understand the allocation algorithm for local variables.

It is generally a little trickier to 'patch' HLL programs, so you may have to content yourself with finding a problem, changing the source, compiling/linking, and then debugging some more.

Being able to establish breakpoints in programs was a major step forward in debugging. Mistakes could be worked around (patched) and testing could be continued. Testing strategies could be modified as results showed actual program behaviour. Many times I came to realize that I had misunderstood what the program did, and that my original test data would be useless for finding the problem. This was really a nifty thing. This was better than single-stepping (most machines can execute on the order of a million instructions per second - a lot faster than pushing a button). The overall techniques used are similar to the 11 steps given in part one of this article, but now without the need to mess around putting in halts, or setting hardware switches. The entire operation is controlled from the console device. Programs stop where we ask, registers are displayed for our convenience.

We've really got it made now! If only...

If only this printout wasn't so slow (10 characters per second)!... Where did those registers go? I just dumped a small amount of memory and the registers are on this printout somewhere... If only I could watch the registers/memory locations while the program is running... What's the sum of hex A733 and 15FD where I need to put my next breakpoint?... If only the debugger could read the link map and let me type in names (instead of numbers)... Hmm, it's time to recompile, better fix the source (sure hope there are some cards in the keypunch).

Next month, we'll look at the revolutions provided by:

- The CRT (faster output).
- Disk files and DOSs (faster editing/compiling/linking).
- Time-sharing (everyone always had a machine available).
- Personal computers (similar advantages to time-sharing).
- Multi-tasking (edit source without leaving the debugger).
- Windowing operating systems (see multiple things at once).

Preview of AmigaDOS V1.3

New goodies and some solid improvements

by Chad Innes

This article is a short preview of the changes and additions to AmigaDOS V1.3 from V1.2. This article is based on the "Gamma 4 version", so keep in mind that it is possible the release version of 1.3 could have some minor changes from this version. Also keep in mind that this is not a full or in-depth review of AmigaDOS V1.3, but simply a sneak peek.

Here is an overview of the major V1.3 changes:

- New KickStart that allows you to boot from devices other than DF0:
- New Fast File System for hard drives (increased speed by a factor of four to five)
- The addition of a PIPE: device
- New CLI with a Shell. Includes recall of commands, editing and more.
- A recoverable ram disk is now included (survives warm starts)
- New faster printer drivers with more printers supported
- New faster IEEE math libraries with support for 68881 co-processor
- New CLI commands and changes to some existing CLI commands

Now onto the details: Two of the first things about V1.3 Workbench you will notice is that in the system drawer, "SlowMemLast" has been renamed to "FastMemFirst". They are the same program, and perform the same function - only the name has been changed (to confuse the innocent maybe?). The second thing is that the all-important (and often used?) "Demos" drawer has been moved from the Workbench to the Extras disk. The Workbench disk is so full that some of the keymaps and printer drivers have also been placed on the Extras disk.

• *New KickStart lets you boot from devices other than DF0:*

The new KickStart allows booting from devices other than DF0:. "What devices?", you ask. I have it from a very reliable source that Commodore *is* working on a hard drive controller card that will allow auto-booting. The second item you can auto boot from is the new recoverable RAM: disk (RRD)! Yes, that's right - reboot your Amy from the RRD. This new RRD is known as RAMBO through the halls of West Chester. More on this a bit later. At this time I am not aware of any other auto- boot devices available or under development, but with this new feature, I feel other items will be appearing soon.

• *New FastFileSystem for hard drives*

The FastFileSystem (FFS) is designed for hard disks and/or non-removable devices only. To use this new FFS you have to edit the mountlist for your hard drive *and* reformat your drive before use. A few points of interest: you still must retain a normal AmigaDOS-formatted partition of your hard drive in addition to the new FFS partition. The normal DOS partition can be as small as one cylinder - but it must exist and it must be the first partition of the drive. The second interesting item is that auto-booting is not allowed from a FFS device. So if you plan on upgrading (or adding) an auto-boot hard drive controller, you will have to make a normal DOS partition on your hard drive. This partition must be large enough to hold all needed information to get your Amy running and transfer control to the FFS area of the hard drive. In testing I noticed a very impressive speed increase. It gives the Amiga a new and different feeling, a feeling of speed and power! During my tests I noticed that using 'addbuffers 20' gave an additional speed boost to the FFS. If you have ever used Facc II for your floppies, you have an idea of the kind of performance boost FFS gives your hard drive. My hat is off to CBM; FFS is one of the finest improvements to the Amiga in a long time.

• *The addition of a PIPE: device*

Commodore has finally give us a PIPE: device. This is a standard Amiga device, not a pipe command like the one available in Unix. To use this new PIPE: you need to add the device to your mountlist and then mount the device before use. Here's a sample of how this device is used.

1> run copy df0:s/startup-sequence pipe:sam
[CLI 2]
1> type pipe:sam

Of course this is a silly example, but it shows how the new PIPE: device works.

• *New CLI Shell includes command recall, editing and more*

How about a shell for the CLI! Features include 'Resident', making commands stay in memory (resident) and called instantly without loading from disk. An alias feature has been added. Alias allows you to replace any command or string with an alias of your own. An example would be:

1> Alias cp copy
1> Alias mv rename

You now can use the Unix names for copy and move (rename). This gives you an idea of the powers of the alias command. Another nice item is an enhancement in the way the shell handles the prompt command. Under V1.2 "prompt %n" would print the CLI number (in place of the %n). This still works in the shell, but now if you use "prompt %n %s" you will be shown the CLI number *and* the "%s" will be replaced by the current directory path. Other features of the shell are command line recall using the cursor up/down keys and command line editing using the cursor right/left keys. This means if you accidentally type:

1> SIR df0:devs/printers

Then press return, you no longer need to retype the entire line. Just press the up cursor key and your line is returned. You don't have to backspace over the entire line just to change the "S" into a "D". Just use the cursor keys to move back to the "S", press DELETE, enter "D" and press return.

• *A recoverable ram disk is now included (survives re-boots)*

The new RRD device is a bit different that the old RAM: disk in a few areas. For one thing, it has to be added to your mountlist, mounted *and* formatted before use. This means that you set its size in the mountlist - unlike RAM:, it doesn't grow and shrink based on its contents. Also you can have FastFileSystem and/or normal file system drives. If you plan on being able to reboot from your RRD, it *must* be of the normal file system type. In most respects, the Commodore RRD does not differ in usage from the commercial RRD (VDK:) or the shareware RRD (VD0:).

• *New faster printer drivers with more printers supported*

Enhancements to the printer drivers include: auto centering, support for three types of dithering are supported, colour correction for Red/Green/Blue, printing speed increases between 3 to 20 times faster than those in V1.2, support for printing of HAM pictures and a new version of *Preferences* with additional printer settings. The additions to *Preferences* include two new gadgets (which are actually two screens), Graphic 1 and Graphic 2. Graphic 1 (printer screen #2) has a new print shade selection (Gray Scale). Graphic 2 (printer screen #3) has several options added, Antialiasing ON/OFF, Left Offset, Center ON/OFF, Density 1 through 7, Colour correction of R/G/B, Ordered/Halftone/F-S dithering, Fraction and Integer scaling support, and Width and Height limits.

• *Faster IEEE math libraries with 68881 coprocessor support*

These new math libraries allow for support of the 68881 math co-processor, either via a 68020 board or with special drivers as an add-on item. In addition to the new 68881 support, the math libraries have been rewritten to be much faster than under V1.2 when not using a math co-processor.

• *New CLI commands and changes to existing CLI commands*

For the changes and additions to the CLI commands, refer to the article called "AmigaDOS V1.3 Manual Update", immediately following.

Well, that was the 25¢ tour of AmigaDOS V1.3. In closing, let me remind you that this was only meant to be a preview of V1.3 and not an in-depth review. Good luck with V1.3; if you want to discuss it or just talk Amiga, drop me a note on Berks Amiga BBS, (215) 678-7691 (24 hours, 300/1200/2400 baud).

INTERCHANGE ™

Share objects between Sculpt3D, VideoScape 3D & Forms in Flight

NOW YOU CAN...

- **Use Sculpt 3D to ray–trace VideoScape 3D objects**
- **Do Forms in Flight animations on Sculpt 3D objects**
- **Create VideoScape 3D objects using the Sculpt 3D interface**

Full Intuition interface for all InterChange functions

Object Disk #1 now available. Includes a Sculpt 3D font, plus lots more Sculpt 3D and VideoScape 3D objects!

InterChange master program plus Sculpt 3D and VideoScape 3D Conversion Modules, $49.95. Forms in Flight add–on Conversion Module, $19.95. InterChange Object Disk # 1, $19.95.

This product requires objects from Sculpt 3D and/or VideoScape 3D and/or Forms in Flight. It is not a standalone animation program.

To order, send check or money order. Please include $3.00 postage & handling. MA and WI residents add 5% sales tax. InterChange is a trademark of Syndesis. Sculpt 3D, VideoScape 3D and Forms in Flight are trademarks of Byte by Byte Corporation, Aegis Development and Micro Magic respectively.

SYNDESIS
20 WEST STREET
WILMINGTON, MASSACHUSETTS 01887
617–657–5585

AmigaDOS V1.3 Manual Update

The new DOS commands and the ones that have changed

by Chad Innes

This article is designed to be a guide used in conjunction with the Bantam AmigaDOS V1.2 manual. The information in this update is based on the Gamma 4 version of AmigaDOS 1.3. Slight changes may be made between this version and the actual release version.

The following syntax will be used for all command descriptions:

[1.3] (When applicable) COMMAND NAME

COMMAND FORMAT

Complete description if the command is new for V1.3 or a listing of the changes from 1.2 version.

ADDBUFFERS

ADDBUFFERS <DRIVE>: <NN>

When used with the FastFileSystem, ADDBUFFERS always speeds up disk access, unlike when used with the normal file system. Only directory blocks, not data blocks, are buffered with the FastFileSystem. The memory type used by ADDBUFFERS is specified in the mountlist via the "BUFMEMTYPE" entry.

COPY

COPY [[FROM]<name>] [TO<name>] [ALL] [QUIET] [BUF=BUFFER] [CLONE] [DATE] [PRO] [COM]

BUF = set the number of 512K buffers used during the copy.
CLONE = combination of DATE, PRO and COM.
DATE = copy the creation date also.
PRO = copy the protection bits also.
COM = copy the comment also.

DATE

DATE <date> <time> TO|VER <filename>

DATE no longer requires a leading 0 in the date specification.

DELETE

DELETE <name> <ALL> <QUIET>

DELETE now continues if one file fails during a multiple file delete.

DIR

DIR <DIR> <OPT A|I|D> <ALL> <INTER> <DIRS>

DIR now accepts "A", "I", and "D" for ALL, INTER (Interactive), and DIRS respectively. There is a "COMMAND=" option in interactive mode that allows most commands to be executed. The DIRS option displays directories only.

DISKDOCTOR

DISKDOCTOR <drive>

DISKDOCTOR now checks for enough available memory before starting.

ECHO

ECHO <string> NOLINE

When using the NOLINE option, the line feed is suppressed after printing.

EXECUTE

EXECUTE <command file> <arguments>

EXECUTE checks for the logical name "T:" and uses it if assigned. Otherwise the ":T" directory is used.

[1.3] **FF**

FF -0|-n

FF (FastFonts) speeds up text display on the Amiga screen. The "-0" option turns FF on and the "-n" option turns FF off.

INFO

INFO <device>

INFO now adjusts for longer device names. The device option allows you to obtain information only on the requested device.

INSTALL

INSTALL DRIVE <DF0|DF1|DF2|DF3>: NOBOOT CHECK

Memory used to create the boot block is now cleared before installing

a disk. NOBOOT takes a installed (bootable) disk and makes it unbootable but still an AmigaDOS disk. CHECK, checks the boot block against standard Commodore boot code. The condition flag is set to 0 if standard or no boot code is found. Otherwise the condition flag is set to 5 (WARN).

LIST

LIST <dir|pattern> P|PAT <pattern> KEYS DATES NODATES TO <name> S <substring> SINCE <date> UPTO <date> QUICK BLOCK

New protection bits are now displayed, H - Hidden, S - Script, P - Pure, and A - Archive. QUICK option does not display extra spaces after the filename. BLOCK option displays the file sizes in blocks instead of bytes. Hidden causes a file to not be displayed during a LIST or DIR command. Note that files are only hidden when using Commodore's Shell. Script allows a CLI script file to be executed just like a program (i.e. without using the *Execute* command). For a description of 'Pure', refer to the *Resident* command. Archive is used as an indication of whether a file has been altered or not. Once set, any command that does a read to the file (like *Ed*, or *Copy*) will cause the archive bit to become unset. This indicates modification of the file.

[1.3] LOCK

LOCK <drive>: ON|OFF <passkey>

LOCK allows you to set the write protect status of a FastFileSystem hard drive or FFS partition. This lock remains in effect until turned off via the lock command or the system is rebooted. Passkey allows you to use a four character passkey when locking a hard drive. The same passkey must then be used to unlock the hard drive.

NEWCLI

NEWCLI <window specification> FROM <file name>

The newcli will automatically become a shell if SHELL-SEG has been made resident. If no FROM file name is specified, newcli uses the default file "S:CLI-StartUp" if present.

PROTECT

PROTECT FILE <filename> FLAGS <+- status bits> ADD SUB

PROTECT can be used to add (ADD|+) or subtract (SUB|-) protection bits of a file. Additional protection bits are now available. They include, Hidden (H), Script (S), Pure (P), and Archive (A). Refer to the LIST command for details of Hidden, Script, Pure, and Archive.

[1.3] RESIDENT

RESIDENT <resident name> <file> <Delete> <Add> <Replace> <Pure>

RESIDENT makes a command memory resident. After a command has been made resident, future use of this command no longer requires disk access to load the command. Entering the RESIDENT command with no parameters will display a list of those commands that are currently resident. Only certain commands can be made resident. These commands must be both re-entrant and re-executable. Commands who meet these rules should have the PURE protection bit set. Resident is only available for use with the Commodore Shell.

RUN

RUN <command> [+ <comment>]

RUN will now start background tasks that won't prevent closing of the CLI from which the task was started. This is done by redirecting the output of RUN. When RUN is used to start a new CLI task, RUN uses the "S:CLI-Startup" file to set up the new CLI if present.

SEARCH

SEARCH FROM <name>|<pattern> SEARCH <string> ALL, NONUM, QUICK

SEARCH now returns a 0 if the object is found and a 5 (WARN) otherwise. SEARCH now stops searching when a Control-C is entered. When using the NONUM option, SEARCH does not print the line number(s) of where the string is found. The QUICK option uses a more compact form of screen display.

SETDATE

SETDATE <file> <date>|<time>

Input of SETDATE has been changed so that the output of DATE can now be used. The use of a leading zero when specifying the date is no longer required. Leaving off the DATE argument defaults to the current date (acting like a 'touch' command).

STATUS

STATUS <process> FULL TCB CLI ALL

Negative priority levels are now supported.

TYPE

TYPE <from> TO <name> OPT H|N HEX NUMBER

TYPE now gives a warning if the destination file exists. HEX (H) and NUMBER (N) options have also been added.

VERSION

VERSION <library name|device name> <version #> <revision #>

VERSION can now be used to find the version number of a library, device, or workbench disk. VERSION can also test for a specific version and set flags if the version is greater (newer). If no parameters are used, the version of KickStart and WorkBench currently in use will be displayed.

[1.3] XICON

XICON

XICON is used to execute a CLI script type file from an Icon. A window for output and input is opened before execution of the script file. To use XICON make a "project" icon for a script file first. Then using the Workbench "Info" menu selection, set the default tool to "C:Xicon".

Workbench Musings

What it is and things we can do with it

by Rob Peck

Rob Peck is the author of the ROM Kernel and Amiga Hardware manuals, and the Programmers' Guide To The Amiga (SYBEX, 1987). He can be reached through: DATAPATH, P.O. Box 1828, Los Gatos, CA, 95031-1828.

Workbench is the most visible part of Amiga's User Interface. Workbench is interesting because it is simply an application program that knows how to launch other programs.

It is easy, as a programmer, to create front ends for your programs that can correctly start the program running regardless of whether it is run from Workbench or from the CLI. The correct method of adapting your C language program for either use is discussed briefly here. For more guidance, I provide references to programs that use these techniques. The "official" way to do it is of course covered in the Amiga ROM Kernel Manual, in the Workbench chapter.

If a user *does* happen upon a program that is exclusively designed for one environment or the other, there are public domain utility programs to let him or her adapt the program to the other environment.

In this article, we'll look at Workbench, "the program". I'll talk about some of the routines that the Amiga provides as a programming interface to Workbench. And we'll look at some of the public domain utilities that folks have created to make it easier to interface to Workbench, or to make a transition from Workbench to CLI and vice versa.

There's A Workbench Screen Without Workbench

When the system starts up from scratch, a Workbench screen is created, even though Workbench itself is not present. You can demonstrate this yourself by doing the following - FORMAT and INSTALL a fresh new disk, with nothing more on the disk (especially no Startup-Sequence). Insert this new disk into the internal drive and reset the machine. What you get after reset is a CLI that looks much like any other computer's text entry display. Now shrink that window by using the sizing gadget, and move the CLI window down to expose the title bar. There you find the Workbench screen, but with no icons, no menus - basically no Workbench. The Workbench screen is just the place where Workbench is built.

Under AmigaDOS 1.2, when your Startup-Sequence issues the command 'LoadWB', the actual Workbench code opens what is called a backdrop window on the Workbench screen. Backdrop windows are unique in that no other window can ever be placed behind that window within a screen.

Workbench draws its icons into the backdrop window, and can then use the standard Intuition window handling (window to front, window to back) with any windows that open on the Workbench screen. Because the other windows always remain in front of a backdrop window, even when pushed to the back, the icons therefore always remain behind any windows.

Workbench Receives Messages

The Workbench backdrop window can receive messages about mouse clicks and mouse movements, just as your own window can receive the same kinds of messages. So Workbench can activate a menu, or deactivate a gadget, just by sensing the user's mouse clicks and movements.

The icons that you see on the Workbench screen are Intuition Gadgets that send additional messages to Workbench. Intuition can tell Workbench that an icon was selected, or it was double-clicked, or it was dragged somewhere onscreen and dropped.

Workbench Icons Take Two Forms

Under version 1.1 of Workbench, when an icon was "grabbed", the mouse cursor turned into an object that was supposed to represent the selected icon. That object was a sprite, as is the normal mouse pointer. Version 1.2 of Workbench takes advantage of yet another built-in Amiga graphics function - the "Bob". A Bob is a graphics element that can be used with the built-in animation functions. The icon image is written into a Bob structure and the animation routines are used to move the image around on the screen, to follow the mouse pointer.

While the original icon image is drawn into the backdrop window, thereby remaining behind all other windows, the image of the Bob is drawn directly into the Screen. This makes it possible to drag the Bob icon image anywhere on the screen, passing over any windows, no matter how many there are on the screen.

Workbench Sometimes Locks Out Other Tasks

Any other tasks trying to draw into the Workbench screen while the icon (Bob) is being dragged around are automatically put to sleep (this happens in the Amiga's internal graphics routines) to provide a stable background against which to move the icon. You can see this if you start some dynamic graphics demo program, such as 'Lines' or 'Boxes', then pick up an icon to move it somewhere else - the drawing into those dynamic windows stops until you have finished with the icon.

This is because the system must save the data in the area where the Bob will be drawn, then draw the Bob image, then restore the image when the Bob is to move somewhere else. If the image underneath the Bob was to change while the Bob was sitting still, then the restored image would not match the saved image and you'd be leaving garbage trailing behind the Bob.

Just Another Application Program

Here we are, then. Workbench: an application program just like any other, defining its own operating environment that includes things to do when icons are moved around by the user. Under version 1.2 of Workbench, we can't do things such as "tell Workbench we just created an icon" or ask Workbench to let that icon drop into our own window just as easily as the icon drops into one of Workbench's own windows (a Drawer window, a Disk Window or icon, a Trashcan window or icon). Workbench directly owns those items and knows how to interact with them, but documents no interface (at least currently) to allow external programs to pass information directly to Workbench.

However, Workbench can use icons that we create to launch our programs, and the routines provided in the *icon.library* make it easy for us to create Workbench-compatible icons. So, until Hackbench (a public domain rewrite of Workbench) grows a bit more mature, or until Workbench itself adds program interactive features, we'll have to settle for having the user do most of the Workbench interaction, but use Workbench's facilities to create the stuff the user actually sees.

Let's look at how your programs can recognize whether they were started from Workbench or the CLI, so that the program can run from either environment comfortably. Then let's look at some of the routines that the *icon.library* provides, so that we can see how some of the freely redistributable programs have used them to adapt to Workbench.

How Do We Find Our Arguments

If a program is designed to start either from Workbench or from the CLI, then it has to be able to find its arguments from either environment. For the sake of an example, assume the program name is TEST. If started from the CLI, the user might have entered:

```
1> TEST item1 item2
```

Your standard C program would be constructed with an entry point at 'main', built as:

```
main (argc, argv)
int argc;
char **argv;
{
    /* program begins here */
```

In the example, 'argc' would have a value of 3. And the arguments themselves would be a set of character strings, formatted as though you had declared an array such as:

```
char *argv[] = {
    "TEST",
    "item1",
    "item2"
};
```

So your program can examine either the name by which the program was called (sometimes the same object code can do different things, depending on which name you used to call it), or the names of the arguments, one at a time.

If called from Workbench, 'argv' has no meaning, and 'argc' will be zero. In this case, a pointer generated by your startup code (that every C program is linked with) has an external pointer value called 'WBenchMsg' that points to the arguments that Workbench has assembled for your program. These arguments represent the one or more selections that the user has made from the Workbench (using extended selection to choose additional icons), before either double-clicking on the last one, or selecting OPEN from the Workbench menu. Now the program becomes:

```
extern struct WBStartup *WBenchMsg;

main (argc, argv)
int argc;
char **argv;
{
    if (argc)  /* true if argc not zero */
    {
        RunAsFromCLI(argc, argv);
    }
    else
    {
        UseWBArgs(WBenchMsg->sm_NumArgs,
```

```
        WBenchMsg->sm_ArgList);
    }
}
```

The value in 'sm_NumArgs' is an argument count, just as 'argc' counts the arguments from the CLI.

The value in 'sm_ArgList' is a pointer to the start of an array of Workbench-style arguments. If, for example, the user had selected two icons, one for item1 and the other for item2, then finally selected and double clicked on the TEST icon, here is how the argument list would appear to your program:

```
struct WBArg arg[3];
```

and internally, each WBArg structure consists of a pointer (actually a BPTR, but that doesn't matter here) to a Lock, and a pointer (a normal string pointer) to a name string:

```
struct WBArg {
    BPTR wa_Lock;
    char *wa_Name;
};
```

As with the CLI arguments, the first WBarg in the list (arg[0]) will represent the icon for the program, and the following arguments represent the other icons that the user chose (probably *tool* icons), in the order that they were selected.

If you want to open the file associated with a selected icon, you first must move into the directory that the icon is located in. The actual file associated with the icon must, of course, be in the same directory as its icon. Changing to the directory is done using the 'CurrentDir' DOS library routine:

```
struct Lock *newlock, *oldlock;

newlock = arg[2].wa_Lock;
oldlock = CurrentDir(newlock);
```

Now you can open the file, using 'Open' or 'fopen' or whatever normal C function you choose to use. For example (using the AmigaDOS Open function):

```
struct FileHandle *fh;

fh = Open(arg[2].wa_Name, MODE_OLDFILE);
```

Functions In The Icon Library

On the Workbench disk, the 'LIBS:icon.library' contains a few functions that programs, mentioned later on, use to manipulate the icons. These are:

GetDiskObject	- read an icon
PutDiskObject	- write an icon file
FreeDiskObject	- a follow-up to 'GetDiskObject'

to free the memory that was used to hold the object when it was read into memory.

Again, the nice part about the above functions is that you don't have to know anything about the format of the files themselves. You just manipulate the data structures that you get from 'GetDiskObject' and so on. (In fact, Commodore-Amiga does not document exactly what these disk data structures contain.)

'GetDiskObject' will return a pointer to a 'DiskObject' structure, which defines an icon. One of the fields in this structure is a pointer to the 'ToolTypes' for the icon. Tooltypes are character strings associated with an icon that can be read by the application. The tooltypes for any icon can be easily changed by the user via the *Info* Menu selection in Workbench. They are used to specify information to the program regarding options, settings, etc., and they convey this information in a standard way (explained in more detail below).

FindToolType - searches through an array of strings for a ToolType that matches the one you are looking for and returns either a pointer to the first character of what is bound to that tooltype, or returns a value of 0.

MatchToolValue - after you use 'FindToolType', you can further search the string that it returns to see if that string contains a particular item.

These two functions are useful for icon ToolType work, but you can use them without referring to icons at all if you wish. 'FindToolTypes' operates on an array of pointers to strings, which can be thought of as though defined as:

```
char *tt_strings [] = {
    "FIRSTITEM=string defining first item",
    "SECONDITEM=another string",
    "THIRD=this|that|the.other.thing",
    ""
};
```

The last line in this pseudo-definition is important - a zero-length string is always the last element in a tooltypes array. The presence of the zero-length string tells Workbench and the icon.library routines that this is the end of the Tooltypes. (This was not described in the ROM Kernel Manual.)

To use 'FindToolTypes', you pass a pointer to the first item in the ToolTypes string arrays and a pointer to the item to find. Here is an example, using the above string definitions:

```
value = FindToolTypes(tt_strings,"THIRD");
```

'FindToolTypes' finds something that says "THIRD=". So it returns the address in memory at which the first character beyond the equals sign in the Tooltype string beginning with "THIRD=" is stored. It is functionally the same as if we had said:

value = "this|that|the.other.thing"

The vertical bars have a special meaning to 'MatchToolValue'. When you want to say that an item in the Tooltypes has multiple values, then you put a vertical bar between the various values that it can have. You can then use 'MatchToolValue' to examine the full string to see if any one of the possible values matches a string that you are looking for. Here are results of calling MatchToolValue for the above 'value' example:

result = MatchToolValue(value, "this");
returns TRUE.

result = MatchToolValue(value, "that");
returns TRUE.

result = MatchToolValue(value, "TEST")
returns FALSE.

Programs That Use The Icon Library

Now, if you know that you are started from Workbench, there will be some additional information available to you, either in the icon file, or in the startup-message you received from Workbench. John Toebes, of the Software Distillery, has written a few programs that use this additional Workbench information, and I wanted to mention them. Working with Workbench icons and the icon library is fairly easy. However, if you are particularly interested in this topic, I'd recommend that you examine the source code for John's examples to see exactly how to read, write and interpret icon information.

The complete source for each of the programs mentioned here is on FISH disks numbers 12 and 43. For your convenience, copies of that source are also available on the Transactor Disk for this issue. At the end of this article, I've created a short example that tests two functions: 'ReadInfoFile', that reads an icon file into memory, and 'WriteInfoFile', that writes it back out again. They are extensions to 'GetDiskObject' and 'PutDiskObject', where I am also providing the directory path to the object.

Alternate Icons

Workbench can display a different icon image when the icon is selected as compared to when the icon is not selected. I have seen people create dual image icons for the Trashcan (a closed and open trashcan), for Drawers (a closed and opened drawer) and for program icons, where the image when selected creates a "credits" notice, among others. To attach an alternate icon to an existing icon, you must:

1. Read the icon file and save any pointers of the data structure that you plan to change.

2. Set the icon flags (gg_Flags) to GADGHIMAGE

3. Set gg_SelectRender to point to an Image structure that describes the alternate image, including where to position the left edge and top edge of the image relative to the icon position; how wide and tall the image is; where to find the image's bit map and other Image information.

4. Write the icon file

5. Put things back the way you found them in the icon data structure.

6. Free the icon.

The program *setalt* attaches an alternate icon image from the CLI environment by accepting the names of a primary and a secondary icon to merge together. Both of the icons have been previously created. This makes it handy because it only requires reading the icon files into memory (using the icon.library functions), modifying a few flags, saving some pointers, then writing out the resulting merged icon.

Saving pointer values is necessary since the icon.library routines all allocate memory for the various data structures that each uses. Effectively, any icon file that has been read in with 'GetDiskObject' creates a set of data structures in memory. By saving various pointers created by the icon.library, you can modify the existing pointer values, such as to point to your own images and string arrays, then write out a modified version of the icon file. But following these modifications, we must also restore the original pointers to allow the system to free the same memory that it allocated when 'FreeDiskObject' is called. The sample program printed with this article saves and restores the tooltypes pointer.

Tooltypes Strings

Here is a really under-utilized feature of the Workbench. 'Tooltypes' strings are accessible from the Workbench environment when the user runs the *info.library* (by using the *Info* menu item). Perhaps the Tooltypes strings are not used often by developers because the user can modify the strings and perhaps it is difficult to rely on the information "staying put" as expected. But that aside, I was able to come up with at least one use, and John Toebes came up with another.

I modified a version of the public domain program SpeechToy that reads its own icon file (if started from Workbench) and locates toolstrings that define the default values that it should use. The default values are those that the user set when SpeechToy was last run. This is in lieu of using the system defaults each time. I have strings that describe PITCH = <whatever>, RATE = <whatever>, SEX = 0 (for male) and so on.

John Toebes used this feature in a program called *IconExec* which allows you to create an icon that effectively contains a script of CLI commands to be executed one at a time. The technique used in the program does not follow the "recommended" (Amiga ROM Kernel Manual, Workbench chapter)

form for tool strings (which should be formatted as "SOMEITEM=SOMEVALUE"), but it works just fine and is not explicitly outlawed by the icon.library, so I guess we can expect to use it. John also provided a companion program called *SetWindow* that establishes a default window size for the CLI that *IconExec* creates (in place of a full screen window).

Another public domain program, created shortly after John's *IconExec*, is called *Xicon*. *Xicon* also lets you execute CLI-only commands from Workbench. I spoke to John about that other program and he mentioned that instead of using Tooltypes to store the commands, it uses the file comment field. It too does the job, but I recommend using *IconExec* instead because AmigaDOS's 'Copy' command does not copy the comment field when a file is copied, though it does copy the Tooltypes fields when the icon file is copied.

The Other Way Around

There are some programs that are designed to be launched exclusively by the Workbench, and provide no CLI interface at all. To conserve memory space, some folks would want to launch all programs exclusively from the CLI. The program *WBrun* was created to fill that need. *WBrun*, also by John Toebes, recreates the program environment of the Workbench, without having Workbench itself loaded.

If, as rumoured, it is possible that Workbench may be offloaded to disk for some future revision of the Amiga operating system, this particular program might well be useful for that version. As John points out in the documentation for this program, *WBrun* is less than 3,700 bytes long. If a user wants to run Workbench programs without Workbench, this is one way to go.

Summary

Although it may not be apparent to the casual user, Workbench itself is one of the most visible demonstrations of the multitasking capabilities of the machine. You launch a program by simply double-clicking an icon, and the Workbench is still there, able to interact with you while the other program is happily running in a window somewhere or on its own screen. Workbench can be pulled to the front by using the LeftAmiga-N key (Commodore-N on the Amiga 500 and 2000) combination, or pushed to the rear by the LeftAmiga-M (Commodore-M) once you are finished using it.

The quickie example that I have provided at the end of the article runs only from the CLI (sigh - can't win 'em all).

But I do feel it appropriate that developers support both CLI and Workbench methods for launching their programs to give the Amiga user the greatest flexibility possible. After all, flexibility is one of Amiga's strong points - let's use it to the fullest extent.

```
/* ReadInfoFile is the function from ROM Kernel
 * Manual that reads an icon file from disk */

/* WriteInfoFile is its created counterpart -
 * no such example in RKM so made it up here */

/* ReadInfoFile is modified to remove typographical
 * errors that I found while trying to make the
 * example work correctly. Also, instead of using
 * the standard Workbench locks, I have converted
 * this to a tool that uses standard AmigaDOS locks
 * on the object or its directory and passing this
 * to the Workbench calls.
 */

/* wbenchdemo.c */

#include <exec/types.h>
#include <exec/memory.h>
#include <libraries/dosextens.h>
#include <workbench/workbench.h>
#include <workbench/icon.h>

/* ReadInfoFile:
 * input  = directory and name of the .info file to be read.
 * returns a pointer to a DiskObject structure if success,
 * or NULL if not.
*/
LONG IconBase;
extern struct Lock *Lock();

struct DiskObject *
ReadInfoFile (icondir, filename)
char *icondir;                      /* which DIR is icon located in? */
char *filename;                     /* what is the name of this icon */
{
   LONG olddir;
   struct Lock *lock;

   struct DiskObject *diskobj;

   diskobj = NULL;                  /* assume failure */

   lock = Lock(icondir, ACCESS_READ);
   if(lock == NULL)
   {
      goto end;                     /* Cannot find directory */
   }
   olddir = CurrentDir(lock);       /* Move to dir where icon is */

   diskobj = GetDiskObject(filename); /* Only then can read it */

   /* Could give complete pathname to get there too
    * I guess, but it seems easier this way. */
```

```
   CurrentDir(olddir);                /* return to original dir        */
   UnLock(lock);                      /* frees the lock on the icon dir */
end:
   return(diskobj);                   /* if successful, returns a pointer
                                       * to a memory area allocated by
                                       * the icon.library... remember to
                                       * FreeDiskObject when done with it. */
}

/* WriteInfoFile.c */

/* input = directory where to write and name of file to write
 * disk object pointer,
 *
 * output = success = 0 if it happened ok, (nonzero) if not.
*/

long
WriteInfoFile (icondir, filename, diskobj)
char *icondir;
char *filename;
struct DiskObject *diskobj;
{
   LONG olddir;
   struct Lock *lock;
   long status;

   status = -1;   /* assume failure */
   lock = Lock(icondir, ACCESS_READ);
   if (lock == NULL)
   {
      goto end;   /* cannot lock target directory */
   }
   olddir = CurrentDir(lock);  /* move to target directory */
   status = PutDiskObject(filename, diskobj);
   if (status == 0)
   {
      status = IoErr();   /* let AmigaDOS say why if possible */
   }
   CurrentDir(olddir);
   UnLock(lock);
end:
   return(status);
}

/* *************** */

/* DEMO PROGRAM RUNS ONLY FROM ---> CLI <--- so that it can print
 * error messages to the CLI that started it.  The format of the
 * command line is:
 *
 * fixicon <dir> <name>
 *
 * where <dir> is the directory where the icon resides
 *       <name> is the root name (without .info added on) of the icon
 *
 * Demo does the following:
 *
 *   reads the icon file (if possible)
 *
 *   DESTROYS old tooltypes array and substitutes a new one.
*/

/* fixicon.c */

main (argc, argv)
int argc;
char **argv;
{
   struct DiskObject *dj;
   long i, success;
   char **tt;
   char *newarray[2];

   newarray[0] = "TOOLTEST=successful";
   newarray[1] = ""; /* zero length string marks end. */

   IconBase = OpenLibrary(ICONNAME,1);
   if (IconBase == NULL) return(NULL);  /* cannot find library */

   dj = ReadInfoFile(argv[1], argv[2]);

   if (dj == NULL)
   {
      printf("ReadInfoFile failed!\n");
      exit(10);
   }
   else
   {
      tt = dj->do_ToolTypes;

      for(i = 0; ;i++)  /* until zero length string found */
      {
         if(strlen(tt[i]) == 0)
            break;
         else
            printf("Toolstring is: %s\n",tt[i]);
      }
      tt = dj->do_ToolTypes;
      /* save old tooltype array pointer */

      dj->do_ToolTypes = newarray;
   }
   success = WriteInfoFile( argv[1], argv[2], dj)

   printf("Value of success is %ld and ioerr() is %ld\n", success, IoErr());

   dj->do_ToolTypes = tt;  /* restore before FreeDiskObject */

   FreeDiskObject(dj);
   CloseLibrary(IconBase);
}
```

Info About .Info

An exposé of the mysterious icon file

by Betty Clay

Betty Clay is a retired mathematics teacher, and a student of Commodore computers. She can be reached via CompuServe (74145,657), QuantumLink (bjc), or by mail at 1322 South Oak Street, Arlington, Texas, 76010.

Editor's note: *this is an in-depth look at the make-up of the ".info" files that define icons. Why do you care what's in these files? Normally, you'll use the system routines for manipulating Icons (GetDiskObject and PutDiskObject), so the specifics of how the file goes together aren't extremely important. That's the theory, but we all know that understanding things at a deeper level always makes problem-solving easier, even if it just means you can do a quick hack to some binary data instead of writing a program to make the same modification. In general, it's safer to use recommended system routines to get things done, but it's always been a Transactor tradition to reveal in detail how things work, just to remove every possible layer of mystery that can potentially block you in your programming efforts. So, in keeping with tradition, we present the Transactor-ized version of Betty's look at Icon files.*

The first thing one learns about the Amiga - and the easiest thing to learn - is to point the mouse at an icon and click. Surely nothing is more fundamental to the standard user interface of this machine. But have you tried to decipher the files that contain those Icons? The documentation is scattered over many areas of the *ROM Kernel Manual*, and I found it rather hard to follow. The best help comes from using a combination of the "Intuition/intuition.i" (or ".h"), and the "Workbench/Workbench.i" (or ".h") files. I found the assembler interpretations much more understandable than those in the C language. Determination and a good imagination were also quite useful.

Shall we take a tour of the Workbench1.2 disk and have a look?

The ".info" file

The first thing I discovered about the .info files is that they are not all alike. Each type of file has its own format. Our tour begins with the file that has no real name except just ".info". This file has a completely different format than the other .info files - its main purpose seems to be to list the names of the other .info files in a drawer or the root directory. Here is a hex dump of the ".info" file from a Workbench1.2 disk that is virtually unchanged from the original. (I added one very short Notepad file to the disk in order to have a project .info file to look at.)

```
0000: F34C0012 00000E32 00000568 0000020C
```

The format of the .info file is not documented anywhere, and so it is hard to do more than guess about the purpose of these first few bytes. The information would not be of much use anyway, since the format is subject to change with future revisions (which is why Commodore-Amiga hasn't documented it in the first place). The rest of the file's contents are more obvious, though: A list of the other Icons in this directory, all in very readable ASCII, with a linefeed ($0A) after each name. The RKM says this is "where our children hang out."

```
0010: 45787061 6E73696F 6E0A436C 6F636B0A
0020: 50726566 6572656E 6365730A 434C490A
0030: 44656D6F 730A5574 696C6974 6965730A
0040: 53797374 656D0A45 6D707479 0A547261
0050: 73686361 6E0A
```

Here is the ASCII equivalent of the above bytes (with periods in place of the linefeeds that separate each string):

```
Expansion.Clock.Preferences.CLI.Demos.
Utilities.System.Empty.Trashcan.
```

The .info file has an extremely simple structure. There is no need for any graphics, because it displays nothing on the screen; it has no flags, and consists almost entirely of text.

The "xxxx.info" files - those that represent Drawers, Tools, Disks, Projects, and Trashcans - have some structures in common, but there are variations according to type. For each of them, the file begins with a 'DiskObject' structure. Drawer, Disk, and Trashcan Icons have a 'DrawerData' structure next, which defines the Window that is opened when the Icon is double-clicked. Following that is an 'Image' structure to define the appearance of the Icon, followed by the actual graph-

ics data for each bitplane in the image. Icons that have two images - for their selected and non-selected states - have a second Image structure and bitplane data after this. Finally, the 'Default Tool' and 'ToolTypes' strings follow, if any. This is the general form for standard Icon files in the current Workbench release.

A walk through Disk.info

Let's look next at the "Disk.info" file. This file is the one that displays the disk icon on the Workbench screen. When you select the disk icon by double-clicking, this file will open the Workbench window into which the various root directory icons (usually drawers) will be placed. Here is the assembler code that describes the information in the DiskObject structure:

```
Structure DiskObject,0
   UWORD do_Magic        ;this is always $e310
   UWORD do_Version      ;1 for current WorkBench version
   STRUCT do_Gadget, GG_SIZEOF ;copy of gadget struct
   UWORD do_Type         ;Types are one of below:
         ; WBDISK=1
         ; WBDRAWER=2
         ; WBTOOL=3
         ; WBPROJECT=4
         ; WBGARBAGE=5
   APTR do_DefaultTool   ; Tool (program) for this file
   APTR do_ToolTypes     ; arguments to program
   LONG do_CurrentX      ; normal X position of Icon
   LONG do_CurrentY      ; normal Y position of Icon
   APTR do_DrawerData    ; describes Drawer's Window
;The final two are used for Tools only:
   APTR do_ToolWindow    ; for program's I/O
   LONG do_StackSize     ; default is 4K, set more if needed
```

Notice that the do_Gadget structure is contained in the DiskObject structure. Icons are actually Intuition Gadgets that Workbench manipulates. Here is a Gadget structure:

```
STRUCTURE Gadget,0
   APTR  gg_NextGadget    ; to link into the system list
   WORD  gg_LeftEdge      ; from left edge to this icon
   WORD  gg_TopEdge       ; from top of screen to this icon
   WORD  gg_Width         ; pixel width of the icon
   WORD  gg_Height        ; how many lines high?
   WORD  gg_Flags         ; highlight or alternate image?
   WORD  gg_Activation    ; what activates the gadget?
   WORD  gg_GadgetType    ; icons are BOOLEAN
   APTR  gg_GadgetRender  ; ptr to image data
   APTR  gg_SelectRender  ; ptr to the alternate image
   APTR  gg_GadgetText    ; ptr to text (not used for Icons)
   LONG  gg_MutualExclude
   APTR  gg_SpecialInfo
   WORD  gg_GadgetID
   APTR  gg_UserData
```

GadgetRender is supposed to point to an Image structure defining the Gadget's appearance, but the pointer seems to have no significance within the file, since the Image structure is in a known position. The SelectRender field, however, is significant in that if it is non-NULL, there is a second Image structure along with its bitplane data following the first; this means that the Icon will have different imagery when it is selected, instead of just being highlighted. Other than this, there is not much to be gained by modifying the Gadget structure; you can't add GadgetText, for example, or even affect the position of the Gadget (that is determined by do_CurrentX and do_CurrentY). The only fields that seem toaffect the actual Icon are Width, Height, Flags, and SelectRender.

Perhaps you noticed that the DiskObject structure contains a pointer to the DrawerData structure. This pointer is used internally, but doesn't seem to relate to the position of the DrawerData structure in the file. It doesn't matter, though, since the DrawerData structure (present only for Disk, Drawer, and Trashcan icons) always follows immediately after the DiskObject structure. DrawerData describes the Window that is opened when the Disk Icon is double-clicked:

```
STRUCTURE DrawerData,0
   STRUCT dd_NewWindow, DD_SIZEOF;
   LONG dd_CurrentX;
   LONG dd_CurrentY;
```

The first thing in the DrawerData structure is a complete NewWindow structure. Anyone who has written a program to open a Window on the Amiga should be familiar with this structure. In this case, it defines the window that will be opened when the Disk Icon is double-clicked. As with the Gadget structure, changing the NewWindow structure to your own specifications has limited effects. Perhaps the most interesting one is changing the DetailPen and BlockPen fields to have the window appear in colours other than the standard ones (e.g. orange and black instead of blue and white).

Following the DrawerData structure is the Image structure that contains the bitmap for the icon. Here are its contents:

```
STRUCTURE Image,0
   WORD  ig_LeftEdge      ; X offset from Gadget left side
   WORD  ig_TopEdge       ; Y offset from Gadget top
   WORD  ig_Width         ; pixel width of the image
   WORD  ig_Height        ; pixel height
   WORD  ig_Depth         ; number of bitplanes used
   APTR  ig_ImageData     ; pointer to image data
   BYTE  ig_PlanePick     ; which planes should be used?
   BYTE  ig_PlaneOnOff    ; how to fill un-picked planes
   APTR  ig_NextImage     ; pointer to next image in list
```

By changing the Image structure, you affect the appearance, size, and position of the Icon's image, but not its Gadget. By making LeftEdge and TopEdge nonzero, for example, you shift the Icon's image from the rectangle in which the user has to click - a confusing situation, which is why LeftEdge and TopEdge are usually zero.

Let us now take a hex dump of the Disk.info file, and tie the assembler descriptions of the structures to the items in the hex dump.

```
0000: E3100001 ........ ........ ........
```

$E310 - do_Magic, a number that identifies this as a .info file

$0001 - do_Version. This is version 1 of the Workbench. If you are working directly with these files, you should make sure that the version is 1 before you assume anything; the format could change in future revisions and create great difficulties for you or your program.

Beginning with the second longword, we find (as indicated in the structure definition above) a standard Gadget structure, though some of the fields are not used for icon files, and are set to null.

```
0000: ........ 00000000 021E0005 00200010
```

$00000000	gg_NextGadget	- set by system, ptr to next gadget
$021E	gg_LeftEdge	- offset from side to this gadget
$0005	gg_TopEdge	- offset from top of window
$0020	gg_Width	- pixel width of icon
$0010	gg_Height	- pixel height

```
0010: 00050003 00010000 C2180000 00000000
```

$0005 gg_Flags, GADGIMAGE ($0004) points to an icon rather than a border, and GADGBACKFILL ($0001 - a flag unique to Icon Gadgets) fills the area surrounding the Icon with the background colour.

$0003 gg_Activation, sets the flags GADGIMMEDIATE ($0002) which sends a message when the icon is selected, and RELVERIFY ($0001), which sends a message when the button is released.

$0001 gg_GadgetType, Boolean, Proportional, String, or Integer. Icons are always Boolean Gadgets (the kind you click on).

$0000 C218 gg_GadgetRender, points to the border or image to be rendered. Icons always use an Image.

$0000 0000 gg_SelectRender, if non-NULL, there is an alternate image for this Icon that will appear when it is selected.

The remainder of the Gadget structure is not used for the .info files, so the remaining fields are all set to zeros:

$0000 gg_GadgetText. This is a long, continued below.

```
0020: 00000000 00000000 00000000 00000000
```

$0000	remainder of the GadgetText pointer, null for an icon.
$0000 0000	gg_MutualExclude bits
$0000 0000	gg_SpecialInfo pointer
$0000	gg_GadgetID (user-definable)
$00000000	gg_UserData pointer

Having completed the do_Gadget portion, we return to the remainder of the DiskObject structure:

```
0030: 013C0001 AB200000 00000000 021E0000
```

$013C do_Type, this is a WBDISK icon. The $01 part of this word is the important part, indicating the Icon type (see definitions in the DiskObject structure above). The lower byte of the word is just for padding, and can be ignored.

$0001 AB20 do_DefaultTool. Pointer to the filename of the default tool for this icon. Example: for a Notepad project, the default tool would be the Notepad. For the Disk.info file, the default tool is always DiskCopy.

$0000 0000	do_ToolTypes - not used in this case
$0000 021E	do_CurrentX - X position of icon
$0000	do_CurrentY - also uses next four bytes.

```
0040: 00050001 B8100000 00000000 00000032
```

$0000 0005	do_CurrentY; Y position of disk icon
$0001 B810	do_DrawerData; Ptr to DrawerData structure

Since this is not a tool, there is no need for the ToolWindow and StackSize fields, so they are set to NULL:

$0000 0000	do_ToolWindow
$0000 0000	do_StackSize

This is the end of the DiskObject structure in the file. Next comes the DrawerData structure (which would not be present if this were a Tool or Project Icon). The first thing in the DrawerData structure is a complete NewWindow structure:

$0032 nw_LeftEdge - distance from the left of the screen to the left side of this window.

```
0050: 00320190 0064FFFF 00000000 0200027F
```

$0032	nw_TopEdge - distance from the top of the screen to the top of this window.
$0190	nw_Width - initial width of the window
$0064	nw_Height - initial height of the window.
$FF	nw_DetailPen - here $FF (-1) sets it to default. This is the colour for the window title, scroll bar borders, etc.
$FF	nw_BlockPen - default colour for the border, scroll bars, etc.
$00000000	nw_IDCMPFlags - none preset in this file.

$020027F nw_Flags - Flags for window Gadgets and attributes. The flags set by this value are WINDOWSIZING, WINDOWDRAG, WINDOWDEPTH, WINDOWCLOSE, SIZEBRIGHT, SIZEBBOTTOM, SIMPLE_REFRESH, NOCAREREFRESH, and WBENCHWINDOW.

```
0060: 00033914 00000000 0000C240 00000000
```

$00033914 nw_FirstGadget - Workbench will attach its own Gadgets
$00000000 nw_CheckMark - not used (for menus)
$0000C240 nw_Title - Workbench will set the window title
$00000000 nw_Screen - used only for a custom screen

```
0070: 00000000 005A0028 FFFFFFFF 00010000
```

$00000000 nw_BitMap - no custom bitmap used here
$005A nw_MinWidth - narrowest width user can size to
$0028 nw_MinHeight - the smallest height allowed
$FFFF nw_MaxWidth - set here to full width of screen
$FFFF nw_MaxHeight - also set to full screen
$0001 nw_Type - a WBENCHSCREEN here

This is the end of the NewWindow structure within the DrawerData; the rest of the DrawerData structure follows:

```
0080: 0000FFFF FFFE0000 00000020 00100002
```

$00000000 dd_currentX - the X displacement of the Icons within this window (The first longword is from the end of line $0070 above). This changes if you move the scroll bars in the Window's border to reveal different parts of a large array of Icons.

$FFFF FFFE dd_CurrentY - the Y displacement of the Icons within this Window.

At this point, we have completed the DrawerData structure that is written to the disk, and now we begin the Image structure that tells the Amiga how to draw the standard disk icon.

$0000 ig_LeftEdge - 0 means line up with Gadget left side (do_Current_X)
$0000 ig_TopEdge - line up with Gadget top (do_CurrentY)
$0020 ig_Width - Icon is 32 pixels wide
$0010 ig_Height - 16 pixels high
$0002 ig_Depth - 2 bitplanes deep

```
0090: 0001B9D0 03000000 00000000 ........
```

$0001B9D0 ig_ImageData - a pointer to the image data (not relevant in file)

$03 ig_PlanePick - selects planes 0 and 1

$00 ig_PlaneOnOff - No un-picked planes to affect

$0000 0000 ig_NextImage - Only one Image for Icons - always NULL

Next comes the data for the two bitplanes that describe the Icon's imagery. Here is how the bits look when represented with dots for zeroes and hash-marks ("#") for ones:

```
..#############....######........
..#############....########......
..#############....##########....
..#############....##############..
..##############################..
..##############################..
..##############################..
..##############################..
..##############################..
..##############################..
..##############################..
..##############################..
..##############################..
..##############################..
..##############################..
..##############################..
.................................
```

This defines bitplane 0, which makes up the low-order bit for the colour selection of each pixel in the image. Now comes bitplane 1 (beginning at the middle of the third longword in line $00D0):

```
00D0: ......................FFFF FFC0C0FF
```

```
############################......
##......###############..####....
##......###############....####..
##......###############......####
##......###############.......###
##......###############........##
##..............................##
##..............................##
##..............................##
##..............................##
##..............................##
##..............................##
##..............................##
##..............................##
################################
```

Notice that the pixels that are turned off in plane 0 are usually turned on in plane 1. If you look at the disk icon on a standard-colour Workbench screen, the ON-bits for plane 1 will be the colour you see for the border (black). On a standard Workbench, colour 0 is blue, 1 is white, 2 is black, and 3 is orange. In other words, the colour you get in any pixel is determined by the corresponding bits in each bitplane as follows:

Bitplane 0	Bitplane 1	Resulting Colour
0	0	0 (Blue)
0	1	1 (White)
1	0	2 (Black)
1	1	3 (Orange)

After the bitmaps, the file ends with the strings that identify the default tools and the 'ToolTypes' (if any):

```
0110: ........ ........ ........ ....5359
0120: 533A5379 7374656D 2F446973 6B436F70
0130: 7900
```

Which, in ASCII, is the null-terminated string:

```
..............SYS:System/DiskCopy
```

This is the program that will be invoked when the disk Icon is dragged to another one in order to copy it.

Variations found in other drawer icons

The "Empty.info" file - a Drawer Icon - is quite similar to the "Disk.info" file, but there are some differences. Notice that the early part of the file differs only in the size and placement of the icon. Also, the flags field here is $0004, indicating that this icon won't be "backfilled", since the entire area occupied by the Gadget is to be highlighted when selected.

```
0000: E3100001 00000000 00820025 0040000D
0010: 00040003 00010001 17000000 00000000
0020: 00000000 00000000 00000000 00000000
```

In the "Disk.info" file, we had the number for a Disk-type Icon at the beginning of line $0030. Here, we have the identifying number for a Drawer (if you look at the DiskObject structure definition, you will notice that a WBDRAWER is file type 2).

The DrawerData starts next, beginning with its NewWindow structure. The only difference here from the Disk.info file are differences in the size and position of the Window.

```
0030: 02F40000 00000000 00000000 00700000
0040: 00180001 F0D80000 00000000 0000011C
0050: 005C00E6 004AFFFF 00000000 0240027F
0060: 0001F14C 00000000 0000AC90 00000000
0070: 00000000 005A0028 FFFFFFFF 00010000
```

The Image structure begins here, and it also follows the same pattern as before, but the Image data following defines a Drawer rather than the disk. You can see the position and size of the Icon described here. Notice that in the last longword on line $0080, the height of the Icon is one less than the height given in the Gadget structure. That is because the designer chose to leave one blank line on the screen between the icon and its name. You are offered this choice in IconEd when you "Set bottom border".

```
0080: 00000000 00000000 00000040 000C0002
0090: 0001A458 03000000 00000000 00000000
```

And here is the bitmap for each plane used in making the drawer icon.

```
00A0: 00003FFF FFFFFFFF FFFC3C00 00000000
00B0: 003C3CFF FFFFFFFF FF3C3CFF FFFFFFFF
00C0: FF3C3CFF FF3FFCFF FF3C3CFF FF0000FF
00D0: FF3C3CFF FFFFFFFF FF3C3CFF FFFFFFFF
00E0: FF3C3C00 00000000 003C3FFF FFFFFFFF
00F0: FFFC0000 00000000 0000FFFF FFFFFFFF
0100: FFFFC000 00000000 0003C3FF FFFFFFFF
0110: FFC3C300 00000000 00C3C300 00000000
0120: 00C3C300 00C00300 00C3C300 00FFFF00
0130: 00C3C300 00000000 00C3C300 00000000
0140: 00C3C3FF FFFFFFFF FFC3C000 00000000
0150: 0003FFFF FFFFFFFF FFFF
```

Since there was no default tool or ToolTypes for this file, the data ended at the end of the bitmap for the second plane.

The "Notepad.info" File - a Tool: Notice first that the tool icon files begin with the same identifying "Magic number." They also follow the same description of the placement and size of the icon.

```
0000: E3100001 00000000 0023000B 00210012
0010: 00050003 00010000 FA680000 00000000
0020: 00000000 00000000 00000000 00000000
0030: 03840000 00000000 00000000 001F0000
0040: 00000000 00000000 00000000 00000000
```

But here, the identification number in line $0030 is $03, (WBTOOL). Note that the StackSize field is NULL - this indicates that the default size supplied by Workbench will be used. Also note that the DrawerData pointer is NULL. There is no DrawerData structure for Tools, so the Image structure comes next, followed by the bitplane data:

```
0050: 00000021 00110002 00027E38 03000000
0060: 00000F07 83C00000 180C0600 00002DF6
0070: FB400000 3FFFFFC8 00003FFC 27CA0000
0080: 3FFB87CA 00003FFE 27CA0000 3FF8E7CA
0090: 00003FE0 07CA0000 338FE7CA 0000263F
00A0: E3CA0000 30FFF0CA 00003FFF FFCA0000
00B0: 00000002 00000AAA AAA80000 00000000
00C0: 00000000 00000000 0F0783C0 0000FFFF
00D0: FFF00000 DE0F07BC 0000C000 00370000
00E0: C003D835 8000C004 78358000 C001D835
00F0: 8000C007 18358000 C01FF835 8000CC70
0100: 18358000 D9C01C35 8000CF00 0F358000
0110: C0000035 8000FFFF FFFD8000 35555557
0120: 80000FFF FFFE0000 00000000 0000
```

This is a rather elaborate icon, and the bitmap looks very different from those we examined above. This icon is 33 pixels wide and 17 lines high. The bitplane data begins in line $0060.

There were no ToolTypes here, so the file ended after the second bitmap for plane 1.

The "Project.info" files: To make this tour as complete as possible, I made a very small file with the Notepad, and named it "Testfile". The data below represents the Icon for the Notepad project which the system supplied for "Testfile."

It is the standard Icon for all Notepad files, of course.

The early part of the file differs from all of the others only in the size and placement of the icon. The pattern remains the same. This gadget is 97 pixels from the left edge and 12 lines from the top. It is 40 pixels wide and 15 lines high.

```
0000: E3100001 00000000 0061000C 00280015
0010: 00040003 00010027 3FD20000 00000000
0020: 00000000 00000000 00000000 00000000
```

In line $0030, we find that the file type is now $04, in keeping with the assignment that says WBPROJECT = 4.

```
0030: 04000026 FBDE0026 FBCE8000 00008000
0040: 00000000 00000000 00000000 00000000
```

As in the others, we have here the image structure, followed by the bitmap for each of the two planes.

```
0050: 00000028 00150002 00273ED6 03000000
0060: 00000000 00000000 3FFFFFCC 00003FFF
0070: FFCF0000 3FFFFFCF C0003803 FFCFF000
0080: 3FFFFFC0 00003803 FFFFFC00 3FFFFFFF
0090: FC003FFF FFFFFC00 3F8400C0 7C003FFF
00A0: FFFFFC00 39008000 7C003FFF FFFFFC00
00B0: 38000040 7C003FFF FFFFFC00 3FFFFFFF
00C0: FC003FFF FE007C00 3FFFFFFF FC003FFF
00D0: FFFFFC00 00000000 00000000 00000000
00E0: FFFFFFFC 0000C000 00330000 C0000030
00F0: C000C000 00303000 C7FC0030 0C00C000
0100: 003FFF00 C7FC0000 0300C000 00000300
0110: C0000000 0300C07B FF3F8300 C0000000
0120: 0300C6FF 7FFF8300 C0000000 0300C7FF
0130: FFBF8300 C0000000 0300C000 00000300
0140: C00001FF 8300C000 00000300 C0000000
0150: 0300FFFF FFFFFF00 00000000 00000000
```

The project does have a default tool, and also has 'ToolTypes' associated with it, which are arguments to the program that the user can change using the Workbench 'Info' Menu selection. The strings for the Tool filename and ToolTypes are found here:

```
0160: 0020576F 726B6265 6E636820 312E323A
0170: 5574696C 69746965 732F4E6F 74657061
0180: 64000000 00100000 00114649 4C455459
0190: 50453D6E 6F746570 61640000 00000E46
01A0: 4F4E543D 746F7061 7A2E2038 00000000
01B0: 1757494E 444F573D 3130302C 2031302C
01C0: 3330302C 31303000
```

Which is, in ASCII:

```
. Workbench 1.2:Utilities/Notepad
.........FILETYPE=notepad
.....FONT=topaz. 8
.....WINDOW=100, 10, 300,100.
```

Each string is preceded by a longword specifying the length of the string, including the null byte that terminates it. The above strings determine the font and window size and position used by NotePad when it is loaded via this project's Icon. Notepad reads these ToolTypes when it is run and acts on them accordingly. (For more on reading Workbench parameters and ToolTypes, see Rob Peck's article in this issue, and the article in the Bits section.)

Transactor Classifieds

Transactor Classifieds are a great way to reach thousands of Amiga enthusiasts inexpensively - just $2 per word! Send your ad copy with cheque or money order to: Transactor Amiga Classifieds, 85 West Wilmot Street, Unit 10, Richmond Hill, Ontario, Canada L4B 1K7.

Programmers Needed: Gold Disk Inc., of Mississauga, Ontario, Canada has immediate openings for two programmers with a minimum one year experience with the C programming language. Candidates should have a working knowledge of INTUITION on the Amiga. Salary Negotiable. Call Lynda: (416) 828-0913.

Free Spirit Software Inc.

ULTRA DOS UTILITIES
MODULE I

Backup those valuable files on your Hard Disk the easy way

- High Speed whole hard drive backup with multi-format feature
- Auto configures to multi-hard drive systems or multi-partitions
- Supports up to 8.5 meg of memory
- Uses a unique double buffered file copy routine for lightning fast copies
- Backs up files
  1) by creation date
  2) with 'Point and click' selection (name, directory, drive)
  3) using 'wild-card' search patterns
- Fully supports Multi-tasking, does not monopolize CPU to achieve its speed. Designed to operate in the 'background'
- Compatible with any Hard drive that follows conventional AmigaDOS file protocol
- 512K Amiga required
- Hard Drive NOT REQUIRED. Ultra DOS Utilities Module I works equally well with dual floppy drives!

Ultra DOS Utilities, Module I is only **$59.95!**

Free Spirit Software, Inc.
905 W. Hillgrove, Suite 6
La Grange, IL 60525
1 - 800 - 552-6777 In IL **(312) 352-7323**
Tech. Asst. **(312) 352-7335**
Selected products available from Financial Systems Software, LTD. in England 0905 - 611 - 463

VISA MasterCard

Random, Or Just Confused?

Is Your Random Number Generator Good Enough?

by David Wood

David Wood is Systems Manager at SMU in Dallas, Texas, where he is taking his doctorate in Image Processing.

Random number generation and testing used to be a topic for the mainframes. Computers affordable to the individual user just did not have the power necessary to do some of the tests. Now with computers such as the Amiga, you have the power. The Amiga has sufficient speed to execute these tests in a reasonable amount of time. The word size is large enough to do realistic math calculations, and there is enough memory to handle relatively large programs.

Even given the right hardware and software support, though, the question arises: Why should you look at random number generators? After all, for almost all kinds of computers there is at least one random number generator supplied with the machine. However, there are several reasons why you should take a closer look at the supplied generator. It may not satisfy all of your needs, or it might be overkill.

Writing your own random number generator gives you ultimate control of the sequence. Many times a custom random number generator may be more appropriate than a standard one. If you are writing games that require many random numbers in a short period of time, the standard random number generator may be unacceptably slow. In that case, you might be willing to accept some questionable statistical properties in your random number stream in exchange for execution speed. On the other hand, some applications needing many random numbers don't really care about speed of execution. For these types of application (for example, large simulations), you will need to rely on tightly-controlled statistical properties. You cannot assume that the built-in random number generator is going to handle all cases. The original designers had to make trade-offs when they created the generator, and the decisions they made may not be acceptable to you.

It is not feasible to describe all the different kinds of random number generators in one article. Instead we will take an in-depth look at 'uniform' random number generators, because their output can be used to generate all other types of random numbers. Numbers pulled from a uniform generator all have the same probability of occurrence. If the generator returns integers from 0 to 9, then each digit has a 10 per cent chance of being chosen.

Warning

As mentioned above, almost every computer has a random number generator, but many of them are not adequate for all tasks [1]. Several years ago, I worked on an Artificial Intelligence project that required mazes. The solution path created by the standard system random number generator had a noticeably regular pattern to it. I did not at first suspect the random number generator - after all, it came with the computer. However, a lengthy search revealed it to be the problem. I needed random integers picked from the range 0 to 3, so I took the random number modulo four. The sequence created was decidedly non-random. It was: 0, 1, 2, 3, 0, 1, 2, 3, ...

Moral: "Never Assume, Check It Out".

History

Possibly the earliest random number table was Tippett's Tables. These tables were created in 1927 and were made of 41,600 digits combined to make 10,400 four-digit random numbers. The numbers were compiled from census reports. The Kadyrov's tables were published in Russia in 1936. These tables contained 50,000 digits and were also created from census data [2]. Fisher and Yates in 1938 took a different approach to making random number tables. They used the 15th to 19th digits of logarithms [2]. Others during the same time period tried to create tables from other sources, such as the telephone directory. Tables eventually fell out of favour, however, because they are clumsy to use and take tremendous amounts of space.

The table that seemed to end it all was done by the RAND Corporation. RAND produced a monumental million random digits using a specially constructed electronic roulette wheel. In 1951, Brown [3, 4] said, "It (the RAND experiment) was an interesting experiment, it fulfilled a useful purpose, and one can do it again that way, if necessary, but it may not be asking too much to hope that ... some ... numerical process will permit us to compute random numbers as we need them."

Categories of Random Number Generation

Many methods have been proposed and used to create random numbers. However, all of the proposed methods fall into six general classes.

The first class is tables of random numbers. Tables are acceptable for manual simulation work and other jobs that are carried out by hand. When used in computers, the tables take up too much space on disk and in memory.

The second class is dice-like methods. Any casino will provide you with dozens of methods of generating random numbers. The plain, ordinary six-sided die is a good example of this class; the roulette wheel is another. Fair dice can be made in the shapes of regular polyhedra to give different numbers of sides, each side having equal probability of coming up. For example, the popular game "Dungeons and Dragons" uses dice with 4, 6, 8, 10, 12 and 20 sides.

A third class of random number generators is hardware devices. Such devices can be directly connected to a computer and the random numbers made available to the users by a subroutine call. The random numbers are generated by reading physical properties like electronic or radioactive noise. Few of these devices were actually installed in computers [5].

The first three classes will give true random numbers. However, they are not useful in a computing environment. The next three classes create random numbers that aren't truly random. For this reason, their output is referred to as 'pseudo-random' numbers. One characteristic of all pseudo-random numbers is that the sequence will eventually repeat. The length of the sequence is called the period.

The fourth class is arithmetic procedures applied to a set of random numbers. This class of random number generation takes a set of generated random numbers created by another method, and attempts to increase the length of the sequence. This increase in the sequence of random numbers from the initial random set is done by an arithmetic rule. One example was a common trick used in the days of paper tape. The computer is connected to two or more punched tape readers, each holding a continuous loop of random numbers (a different set of numbers on each reader). The computer reads each tape and combines the numbers, for instance by summing them modulo P, where P is the number base of the computer [2].

The advantage of this method is that if the lengths of the tapes are relatively prime, then the length of the created random number stream is the product of the tape stream lengths. If the computer has three tape inputs, then three short tapes of length 7, 11, and 13 random numbers each will generate a stream of 1001 pseudo-random numbers. From an input of 31 numbers, over one thousand numbers will be created. The drawback to this method is that the generated sequence is not truly random. By definition, if the loop tape values are known, then all of the generated digits will be known. Even if the loop tape values are not known, they can be deduced by observing a large enough sample.

The fifth class of random numbers comes from arithmetic procedures. Like the fourth class described above, this generator requires a set of initial values. The main difference between the fifth and the fourth class is that the initial values can be chosen arbitrarily, sometimes subject to some constraints. Another difference is that the number of initial values required for the fifth class is typically between one and the low tens, whereas class four usually requires hundreds of seed numbers to generate usefully long random number streams. The fifth class of random number generators is usually represented by a function that looks like:

$$x_n = f(x_{n-1}, x_{n-2}, x_{n-3}, x_{n-4}, x_{n-5}, x_{n-6}, \ldots)$$

Clearly, the fourth class number sequence will eventually repeat, since all inputs eventually repeat. It is not quite so clear that this method will do the same. However, since the digital computer has only a finite number of bits to use to generate the random number sequence, it too must eventually repeat. In addition, if the initial sequence is known and the generating function is known, the generated random number stream is also completely predictable.

The sixth and final class of random numbers are generated from the digits of constants like *pi* and *e* (base of the natural logarithm). These constants are called transcendental numbers, and their advantage is that their digits never repeat. The disadvantage is the difficulty in calculating long sequences of digits. As the number of digits grows, the difficulty of getting the next digit grows more than linearly [6]. This method is extremely interesting from a mathematical viewpoint but it is impractical for computers because of memory and execution speed limitations.

Desirable Characteristics

What should you look for in a random number generator? Three main criteria must be met: a long period, reproducibility, and statistical acceptance. Additional criteria, such as execution speed or size of algorithm, may also be important for an individual application.

Strange as it may seem, the random number generator should be able to create reproducible streams. The same initial values fed into the random number generator should create exactly the same stream. This property is important for debugging. If the program does something unacceptable or unexpected, it is important to be able to reproduce the conditions under which the anomaly occurred as exactly as possible, which includes being able to re-generate its pseudo-random input.

The generated random numbers must pass statistical tests to make sure that they are correctly distributed. For instance, a random number generator that does not return all possible values is not acceptable.

The first three classes, in general, do not generate sequences that are reproducible. This is not a problem with the last three generators. Moreover, by the proper choice of initial values, all three of the last classes can generate sequences with sufficiently long periods.

Statistical acceptability can be shown for classes four and five. Class six, the transcendental numbers, becomes too complex too quickly to be of practical use, since each digit requires an increasing amount of work. Class four algorithms are the simplest, but they require large amounts of memory and can be slow, especially if the values are not all kept in memory.

This leaves us with class five, which is the most often used form. Class five generators can be compact, fast and efficient, and do not carry the memory load that class four generators have. Almost all machines today have one or more arithmetic random number generators available, most often as part of a callable standard system library. For example, the Amiga Lattice C compiler has several random number generators in its library, and AmigaBasic has a random number generator built into the language. The Lattice C and AmigaBasic generators are examples of the class five method.

There is probably an infinite number of class five algorithms. We will look at several kinds that have been used in the past. After a short description of each of the algorithms, we will look at some of the statistical tests used to check the 'randomness' in random numbers. In spite of what you might think about statistics, most of these tests are very simple and obvious once you have seen how they work.

Class Five Algorithms

An early method, called the 'mid-square technique', was proposed by von Neumann in the 1940s. To generate the next random number, take the previous number and square it. Take the middle digits as your next random number. In general, when you square a number, you double the number of digits. As an example, let's say that we are generating pseudo-random numbers from 0 to 100. The initial number should be an even number of digits and the first digit should not be zero. For this example, let's start with the initial seed of 57. The square of 57 is 3249, so your next pseudo-random number would be the middle digits, in this case 24.

This method has many advantages. First, it is fast. It requires only one multiplication, an AND operation to mask off the unwanted digits, and a shift to move the middle digits over. Second, the program code required to implement this method is small and third, It only requires one seed value to operate. It has one serious drawback, however: the mid-square technique can sometimes generate very short periods. To continue with the example, the square of the created pseudo-random number 24 is 0576. The middle section of this number is 57, which is the starting point. The initial seed created a period that is two numbers long. This method can also generate strings of good random numbers and then fall into a small cycle. The initial string of good numbers, called a 'tail', is sometimes hard to detect. A long tail may fool you into thinking that the random number generator has a long sequence of numbers when it doesn't [1].

An early suggestion was made to multiply the previous two pseudo-random numbers together instead of squaring. The middle digits of the result would be used as the next random number. As it turns out, no advantages resulted from this change.

An improvement in von Neumann's method was proposed by von Hoerner, who proposed that in certain cases a constant be added after the original random number had been squared. If the squared number is greater than some constant, say B, then add the constant A to the squared number. If the original squared number is less than B, do not add anything.

The advantages to this is that it is still fast, compact, and it still only requires one seed value to initialize it. With the proper choices for the constants A and B, the method can prevent sequences that end in cycles of length one. Unfortunately, it can still generate sequences with small cycles, so it turns out not to be much of an improvement over von Neumann's method.

Congruential Random Number Generators

Today's computers use congruential generators or variations of this generator. The equation for a general congruential generator is of the form below. The generated pseudo-random number is x_n, and the J previously generated values are x_{n-1}, x_{n-2}, through x_{n-J}. All of the As and the B are constants. P is also a constant - usually the word size of the machine. To get the next pseudo-random number, take the last N generated pseudo-random numbers, multiply each one by a previously determined constant, add them all up, add in another constant for good measure, and take the whole thing modulo P.

$$x_n = (A_1x_{n-1} + A_2x_{n-2} + \ldots + A_Jx_{n-J} + B) \text{ modulo } P$$

The advantages of this equation is that it is still fast and compact, although not as fast or compact as the first set of methods. The best quality of congruential generators is that it can generate long periods of pseudo-random numbers with good statistical characteristics. Of course, this depends on the choices for the constants and the initial seed values. By selecting the correct values, congruential generators can guarantee that the period will be P^{j+1} long. Since this is exponential, extremely long sequences can be created easily.

Additive Subclass

There are several common subclasses of congruential generators. The first is the additive method. Selected past values are added together and then taken modulo P. Each A constant is either 0 or 1. The simplest one of this type is the Fibonacci generator. It simply adds the previous two random numbers and then takes the modulo.

$$x_n = (x_{n-1} + x_{n-2}) \text{ modulo } P$$

The problem with the Fibonacci generator is that it is obvious how the pseudo-random numbers are created just by looking at the sequence. Take a look at this example. You will be able to tell what the value of the modulo P is almost immediately. From an initial seed of 5 the sequence is:

5 5 10 15 25 40 15 5 20 25 45 20 15

You can see several problems from the sequence already. It only returns values that end in 0 or 5 and it certainly does not look random. It also shows that the selection of constants for the generator is critical to its success. Here the values of 5 for the seed and 50 for the modulo make a bad number stream. Tests will be presented later that can be used to confirm this.

Multiplicative Subclasses

The next subclass is the multiplicative method. Here the new pseudo-random number is created by multiplying the old value by some constant before doing the modulo. This method has a problem in that it cannot generate periods as long as the general congruential method.

$$x_n = (A + x_{n-1}) \text{ modulo } P$$

The subclass called the mixed method is the multiplicative method with a constant B added to it. The main advantage of the mixed subclass is that the maximum period can be generated with the proper choice of B.

$$x_n = (A + x_{n-1} + B) \text{ modulo } P$$

Finally, there are miscellaneous generators that use combinations of the above. Most of the practical generators do not use one method but a combination of all of the above. As an example, MacLaren-Marsaglia proposed a practical random number generator that combined several methods. The MacLaren-Marsaglia generator started with a 128 location table in memory that was filled by a multiplicative generator. Using a mixed type random number generator, the MacLaren-Marsaglia algorithm took the seven most significant bits and used them as an index into the table. The number at the location was taken to be the next pseudo-random number. The location was refilled by another multiplicative generator.

Your Turn

By now you should have a good idea what has happened in the past. Do you think that you have a scheme that is better than what has been described? Maybe you want to check out the random number algorithm from the article "Random Number Generation in Machine Language" by Gregory D. Knox in the May, 1987 issue of *Transactor*.

Eventually, the time will come to test your ideas. I will step through the major tests that all random number generators should pass. Keep in mind the repeating sequence 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4 ... as you read through the tests. The repeating number sequence is for comparison purposes and as a warning. You may be surprised to learn that this number sequence will pass some statistical tests perfectly. This is a warning that to test random numbers, you have to use a variety of tests, not just one or two. With that in mind, let's start.

The Statistical Tests

The statistical tests are applied to samples of the random number streams. Ideally, the tests should be selected according to the sensitivities in the application. If you want a fast random number generator for a game, for example, you may not need to do all of the quality checks. The tests covered here are:

- Frequency Test/Equi-distribution Test
- Serial Correlation - Between Digits
- Serial Correlation - Between Every Kth Digit
- Poker Test
- Gap Test
- Runs Up/Down Test
- Runs Above/Below Mean Test
- Kolmogorov-Smirnov Test

From the names of the tests, you can guess what most of them do. Most of the tests take less than 10 lines of C or Basic program code. *(Editor's note: the author's C language implementations of all the tests can be found on the disk for this issue of the magazine).*

One point to remember about statistical tests is that you must generate a sufficient quantity of random numbers to make the test valid. Too few can cause a test to incorrectly fail a generator when it should have passed or, even worse, to pass one that should have failed. How many samples are 'sufficient'? It varies with each test and with the conditions of each test. To be on the safe side, I used 2500 pseudo-random numbers to do my tests. In general, you do not have to worry about having too many samples.

Frequency Test or Equi-distribution Test

The random numbers generated should be uniformly distributed. This test splits the random number range up into equal-sized intervals and tallies the number of random values that fall into each range. If the random number generator is uniform, you would expect the populations in all intervals to be roughly equal.

Here is a sample set of uniform random numbers on the range of 0 to 99. For convenience, let me define my interval to be 10 numbers wide, so I will have 10 intervals.

41	68	89	94	74	91	55	62	36	27
19	72	75	8	54	2	1	36	16	28
18	1	95	69	18	47	23	32	82	53
31	42	73	4	83	45	13	57	63	29

Interval	Count	Interval	Count	Interval	Count
00-09	5	10-19	5	20-29	4
30-39	4	40-49	4	50-59	4
60-69	4	70-79	4	80-89	3
90-99	3				

There is danger in small samples. With this small a sample, all you can say is that it looks like the random numbers are well distributed. Since you cannot really be sure, larger samples are needed.

Serial Correlation - Between Digits

This is a two-dimensional test. This test splits the random number range into equal sized intervals, as before. Two successive random numbers are taken and their intervals are used as coordinates into a square matrix. The matrix location is incremented to keep a count. If the random number generator is uniform, you would expect the matrix to have roughly equal count values over its whole area. This method can be generalized to three, four, five and higher dimensions. As a general rule, the higher dimensions do not tell you much more than you will learn in the one-dimensional (frequency distribution) and two-dimensional cases.

When selecting the random values for the serial correlation test, you have two options - overlapping and non-overlapping. You may use either one; it should not affect the statistics. Overlapping is when pairs of random number are taken in this way; x_n and x_{n-1}, x_{n-1} and x_{n-2}, x_{n-2} and x_{n-3} and so on. Non-overlapping is x_n and x_{n-1}, x_{n-2} and x_{n-3}, x_{n-4} and x_{n-5} and so on.

Serial Correlation - Between Every Kth Digit

This is the same test as the serial correlation between digits with the exception that random number samples are K apart. The value K is called the lag. This test can also be multi-dimensional, with different K values between each dimension.

Poker Test

This test is applied to non-overlapping groups of five random numbers. These groups are checked for combinations, some of which are the same ones that are of interest in poker. The probabilities of the different combinations are easy to calculate. Some of the more common combinations for groups of five numbers are in the following table. The probabilities for both bases are exact.

Poker Hand		Base 10	Base 2
Bust	(abcde)	.3024	-
Pair	(aabcd)	.504	-
Two pairs	(aabbc)	.108	-
3 of a kind	(aaabc)	.072	-
Full House	(aaabb)	.009	.625
4 of a kind	(aaaab)	.0045	.3125
5 of a kind	(aaaaa)	.0001	.0625

This test may be applied to groups with an arbitrary number of digits. Groups of four digits are popular and the probabilities are easy to calculate.

Gap Test

Two different types of gap tests exist. The simpler one works with the digits of the random number stream, and the second works with the values.

In the first test, a gap is defined to be the distance between like digits. For example, if the random number stream begins with 0 2 3 2 3 0 0 1 3 2 1 0 1, then the first gap for digit 0 is four long. From this, you can see that the gap size can range from 0 (digits are neighbors) to infinity. If the gap size is too big, the random number generator has failed the test already. Still using the digit 0 as an example, suppose you collect all of the gap sizes. Now you need to know if the sequence has a problem. Sum up all of the gaps that you have found and divide it by the total number of gaps. On a sample set of data, these statistics were collected:

Digit	Gap length 0	1	2	3	4	5	6	7	8	>8	Total gaps
0	8	7	6	3	2	1	1	1	1	0	30
Calculated probabilities											
	.2667	.2333	.2000	.1000	.0667	.0333	.0333	.0333	.0333	.0000	1.0
Predicted probabilities											
	.2500	.1875	.1406	.1055	.0791	.0593	.0445	.0334	.0250		1.0

If you mentally plot the data from the calculated probabilities and the predicted probabilities, you will find that the two curves have the same basic shape. The calculated probability curve overshoots the predicted, especially for gap lengths 1 and 2, but this may not be an indication of a problem. This was done with a small sample size, so use more random number samples to do a better check on the generator.

To calculate the predicted probability, use the following equation. The variable r is the size of the gap and b is the number of intervals that you are using.

$$\text{Prob}(r) = \left(1 - \frac{1}{b}\right)^r * \frac{1}{b} \qquad (r = 0, 1, 2, 3, 4, \ldots)$$

The second type of gap test checks for the distance between local maxima in the random number sequence. A local maximum is defined to be a number surrounded by a smaller number on each side. The gap is the distance between one local maximum and its neighbor. This test should also be repeated for the local minima.

The sample random number stream 0 2 3 2 6 8 9 4 1 3 2 8 0 1 has its first local maximum at the first 3 and its second local maximum at the 9. The distance between the gaps is 5 digits

long, including the local maxima themselves. The first local minimum is at the second 2, and the second local minimum at the second 1, which makes this gap length six digits long. Since the maxima (minima) digits are included in the length, the smallest gap size that you can get will be three digits. Using the same procedure as the first gap test, you collect all of the gap sizes. Sum up all of the gaps that you have found and put the result into the total gaps column. Divide the individual gap counts by the total number of gaps to get the calculated probability. On a sample set of data, the gap test for local maxima was used. The format is the same for the minima.

Digit	Maxima Gap length 3	4	5	6	7	8	9	>9	Total gaps
Count:									
0	16	13	7	3	1	0	0	0	40
Calculated probabilities									
	.4000	.3250	.1750	.0750	.0250	.0000	.0000	.0000	1.00
Predicted probabilities									
	.4000	.3333	.1714	.0667	.0212	.0057	.0013		1.00
Count * Gap Length									
	48	52	35	18	7	0	0	0	160

To calculate the predicted probabilities, use the following equation. The variable r is the size of the gap. Remember to include the local maxima or minima in the length of the gap.

$$\text{Prob}(r) = 3 * \left(\frac{2^{(r-1)}}{r!} \right) * \frac{r-2}{r+2} \quad (r = 3, 4, 5, 6, \ldots)$$

Multiply the gap length by the number of gaps with that length. Sum these numbers and divide by the total number of gaps. This is the mean (average) gap length and, for uniform random number generators, should be near 4.00. In the ideal case, the gap length average is precisely 4. In the example, the mean is 160 / 40 = 4.0 exactly, by chance.

Runs Up/Down Test

This test measures the number of increasing or decreasing sequences that are contained in the random number sequence. This test is similar to the gap test in that counts of sequence lengths are kept. To check for a runs up sequence, start at the first random number and count the random numbers until the current random number is less than the previous. This is called a monotonically increasing sequence. Do the same for monotonically decreasing sequences. To make this explanation a little clearer, here is an example. The numbers that are 'runs up' are indicated by plus signs above the sequence and 'runs down' by minus signs under the sequence. For example, the sequence:

$$41^{+}68^{+}89^{+}94_{|}74^{+}91_{|}55^{+}62_{|}36_{|}27_{|}19^{+}72^{+}75_{|}08^{+}54_{|}02_{|}01^{+}36_{|}16^{+}28\ldots$$

has one sequence of length three for both the 'runs up' test and the 'runs down' test. The statistical information is collected the same way as for the gap test.

Digit	Run length 1	2	3	4	5	6	>6	Total runs
Runs up								
0	5	1	1	0	0	0	0	7
Runs down								
0	4	1	1	0	0	0	0	6
Sum of runs up/down								
0	9	2	2	0	0	0	0	6
Calculated expected run length								
	8.4167	3.4333	0.9250	0.1889	0.0312	0.0043	0.0043	

The expected number of each run length can be calculated from the following equation. The equation is long but easy to program. The variable N is the number of random numbers in the sequence and the variable r is the length of the run.

$$\text{Expected length}(r) = 2 * N * \frac{r^2 + 3r + 1}{(r+3)!} - 2 * \frac{r^3 + 3r^2 - r - 4}{(r+3)!}$$

Runs Above/Below Mean Test

This test, also called the sign test, checks for 'runs above' and 'runs below' the mean (average). In this test, you are looking for sequences in the random number stream that are above the mean as well as below. Count each run length for the 'runs above' test and calculate the percentage of each. Repeat this for the 'runs below' test. As an example, take the same random number sequence as above, which has a mean of 47.4, and perform this test on it.

$$\begin{array}{cccccccccccccccccccc} + & + & + & & + & & + & & & & + & + & & + & & & + & & + & \\ 41 & 68 & 89 & 94 & 74 & 91 & 55 & 62 & 36 & 27 & 19 & 72 & 75 & 08 & 54 & 02 & 01 & 36 & 16 & 28 \ldots \\ & & & - & & - & & - & - & - & & & - & & - & - & & - & & - \end{array}$$

Digit	Run length 1	2	3	4	5	6	7	8	>8	Total runs
Runs above										
0	1	1	0	0	0	0	1	0	0	3
Runs below										
0	2	0	1	0	1	0	0	0	0	4
Calculated probabilities for runs above										
	.3333	.3333	.0000	.0000	.0000	.0000	.3333	.0000	.0000	1.00
Calculated probabilities for runs below										
	.5000	.0000	.2500	.0000	.2500	.0000	.0000	.0000	.0000	1.00
Predicted probabilities										
	.5000	.2500	.1250	.0625	.0313	.0156	.0078	.0039	.0039	1.00

The equation to check the expected run lengths is much simpler for this test. The probability of a run of length r is given by:

$$\text{Prob}(r) = \left(\frac{1}{2} \right)^r$$

It looks like this random number sequence may have a problem with too many numbers above the mean. The 'runs above' test has a sequence of length 7 and the probability of that occurring is small. This points out (again) the dangers of small samples. The overall stream may be acceptable but this short

segment certainly is not. It also points out the need for multiple tests on the same random number sequence. This short string of numbers passes the up/down test but fails the above/below.

Kolmogorov-Smirnov Test

This is the most difficult test to understand but it is also one of the most powerful. All random number sequences have a characteristic curve called the cumulative distribution function, CDF for short. The X axis of the CDF is all of the possible random numbers that can be generated. The Y axis is the probability that the random number generated is less than or equal to the number you picked from the X axis.

Suppose you are generating numbers uniformly between 2.0 and 4.0 and you need know what is the probability that the random number will be less that 3.5. The CDF curve will tell you that. You can read directly from the curve that the probability is 75%.

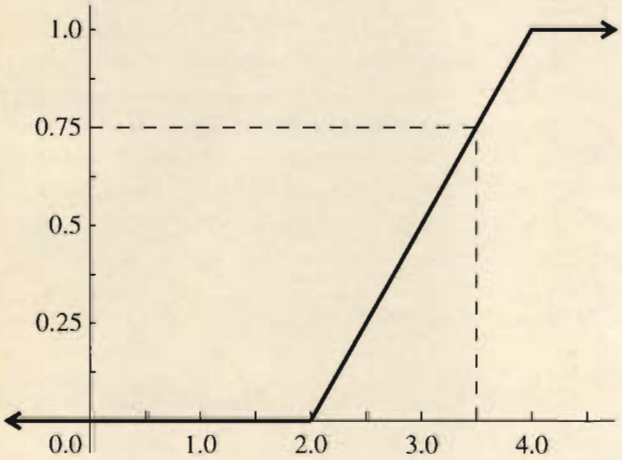


Figure 1. Cumulative Distribution Function (CDF)

As a check to see if the CDF is correct, there are three conditions that always must be met. First, the CDF must be zero for all values smaller than the smallest value from the random number generator. Second, the CDF curve must be one for all values greater than the largest number from the random number generator. Finally, the curve must always be either level or increasing. If it is not monotonically increasing, it cannot be a cumulative distribution function. It must always start at zero and end at one. These conditions make sense if you think about them. There can be no generated random numbers smaller than the bottom of the range, so the probability of this occurring is zero. Likewise, all random numbers will be smaller than any number larger than the maximum.

This curve is an integral part of the Kolmogorov-Smirnov test. This test compares the distribution from the random number sequence to see if it matches the distribution you intended. Uniformly generated random numbers have a simple triangular distribution, with the base being the range of the numbers to be generated.

Selecting the Critical Value

The following two paragraphs are important to understand if you want to correctly apply the Kolmogorov-Smirnov test. The table referred to next is the "Kolmogorov-Smirnov Critical Values" table, which can be found at the end of this article.

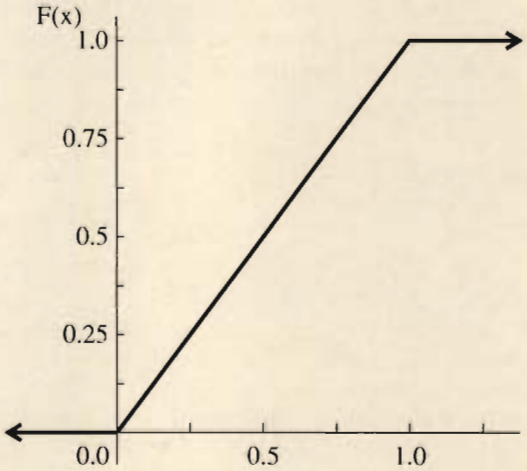


Fig. 2a. F(x) - Ideal uniform cumulative distribution function

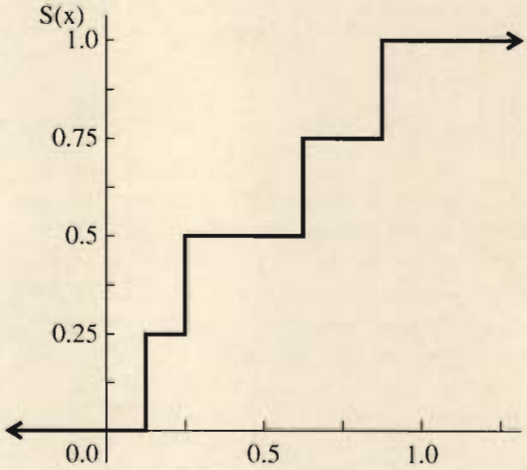


Fig. 2b. S(x) - Sample cumulative step-function using 0.125, 0.25, 0.625, and 0.875 as the random numbers

Before you can choose a critical value from the table, you need to know the meaning of 'level of significance'. Let x be an arbitrary point on the X axis of the cumulative distribution function (CDF). Define function F(x) to be the ideal uniform CDF and function S(x) to be the cumulative step-function of the random numbers.

A step-function is defined to be S(x) = k / N where k is the count of random numbers less than or equal to x and N is the length of the sequence. (See Figure 2 for examples of F(x) and S(x).) The 'level of significance' is the probability that, for all possible values of x, the absolute value of the difference between F(x) and S(x) is greater than or equal to the critical value [8].

$$Z = \text{Maximum of } | F(x) - S(x) | \text{ for all values of } x$$

$$\text{Level of Significance} = \text{Prob } (Z >= \text{Critical Value})$$

Choosing the level of significance involves a tradeoff: the larger the value, the stricter the requirements to pass, but also the better chance to fail a legitimate stream. On the other hand, the smaller the level of significance, the less chance there is to fail a good stream but a better chance to pass a bad stream from a generator with a similar CDF. A level of significance of 0.10 means there is a 10 percent chance that your set of random samples will fail the test even though they came from a uniform generator. Your choice of level of significance is important. For most purposes, a level of significance around 5% to 10% (0.05 to 0.10) is adequate.

If the calculated distribution statistic D is less than the table's critical value, there is no detected difference between the true distribution and your created sequence. This is not to say that the sequence *is* uniformly distributed, but merely that no differences were large enough between the ideal and your data to fail the test.

The Five Steps

The Kolmogorov-Smirnov test has five basic steps. The variable D is the distribution, N is the length of the random sequence, and i is a loop index. Array RV(i) is the ith sorted random variable.

The first step is to sort the random numbers from the smallest to the largest. Duplications should not be removed when sorting. Store the sorted numbers in the RV array.

The second and third steps of the test calculate the distribution value D. Calculate D^+ and D^- by using the following equations. D^+ is the largest deviation of the random number sequence above the ideal uniform random number CDF curve. Similarly, D^- is the largest deviation below the ideal uniform CDF. The distribution value D is the maximum of these two values.

$$D^+ = \underset{1 \le i \le N}{\text{MAX}} \left(\frac{i}{N} - RV(i) \right)$$

$$D^- = \underset{1 \le i \le N}{\text{MAX}} \left(RV(i) - \frac{i-1}{N} \right)$$

$$D = \text{MAX}\ (D^+, D^-)$$

For the next step, look up the critical value from the Kolmogorov-Smirnov table. The column headed by the caption 'Sample Size' is the length of the sequence that you are testing. (Variable N is looked up in this column.) A small version of this table can be found at the end of this article; most general statistics books have a larger version. The values across the top correspond to the level of significance that you want in your sequence. Choose the appropriate critical value from the table that suits your application. In general, you should use more than 35 random values when testing, so you will have to calculate your critical values using the equations at the bottom of the table. However, the example below will use five values so it will need the table portion.

The fifth and final step is to check the distribution statistic D that you calculated from the random number stream and compare it to the table's critical value. If D is greater than or equal to the critical value, then the data stream fails as a uniformly distributed sequence.

Kolmogorov-Smirnov Example

The explanation of Kolmogorov-Smirnov is rather lengthy; an example should clear up some of the confusion. For this test, take the random number stream to be five numbers long. The numbers are supposedly uniformly generated from the range zero to one. The five numbers are 0.45, 0.81, 0.14, 0.06, and 0.93, in that order.

Let's say that we want a level of significance of 5%, so we will use the $D_{0.05}$ column. First, sort the numbers in ascending order and calculate the values for the equations. This will create the following table.

i	1	2	3	4	5
RV(i)	0.06	0.14	0.45	0.81	0.93
i / N	0.20	0.40	0.60	0.80	1.00
i / N - RV(i)	0.14	0.26	0.15	-0.01	0.07
(i - 1) / N	0.00	0.20	0.40	0.60	0.80
RV(i) - (i - 1) / N	0.06	-0.06	0.05	0.21	0.13

Scanning across the third row for the maximum will give D^+ the value 0.26. D^- is the maximum on the fifth row, which is 0.21. Therefore the distribution D is the maximum of the two or 0.26. Now look up the critical value in the Kolmogorov-Smirnov table for $D_{0.05}$ (5% level of significance) and N equals 5. The value should be 0.565. Since the calculated value 0.26 is less than the critical value of 0.565, there is nothing to indicate that the random numbers did not come from a uniform random number generator.

For Further Exploration

There are many books on the subject of random number generation. The books written from the statistical point of view generally start at the level of the clouds and go up from there. Unless you are well versed in statistics, it is generally not a good idea to start there for more information. A better place for introductory material on random number generation can be found in better books on simulation. Simulations, especially large simulations, are sensitive to the random number streams, so these books will usually have a appendix or a chapter or two about random number generation [7]. These books do not always assume a high level of statistical knowledge. Another

place for introductory material is the bibliography at the end of this article. This article only touches the surface of the subject; there are many other aspects that I do not have space to cover. Happy explorations.

Conclusion

This article presents a high level overview of what needs to be done to certify a random number generator as acceptable. There are many tests that need to be performed on a generator before you can be confident that it will suit your needs. Also, just because a random number routine is supplied with the hardware or software, you cannot assume that it will meet the needs of all people all the time.

At the risk of repetition, run all of the tests on your random number sequence. One or two tests can be worse than no tests at all. Too few tests may give false confidence in the randomness of the sequence. Even if a set of random numbers passes all of the tests, it is still not a guarantee of randomness. It is always possible that an underlying pattern will be missed. There are many tests that check other aspects of randomness. See the end of this article for references to other authors.

Thorough testing is a lot of work but it is rewarding once you find a generator that meets the needs of your application.

Annotated Bibliography

[1] The article "Random Number Generation on Microcomputers", by Doan T. Modianos, Robert C. Scott, and Larry W. Cornwell is well written and informative. It does not mention the Amiga because it was printed before the Amiga was released. The article can be found in the July-August, 1984 issue of Interfaces, pages 81-87.

[2] The historical information came from many different sources but one book seemed to sum it up the best - Random Number Generators, written by Birger Jansson in 1966. This was a good source of information because it started with the basics and built from there. The book was printed in Stockholm by Victor Pettersons Bokindustri Aktiebolag.

[3] G. W. Brown (1951), "History of RAND's Random Digits-Summary", Monte Carlo Method (A. S. Householder, editor), National Bureau of Standards, Washington, D.C., pages 31-32.

[4] Additional information on the RAND project was provided by The Handbook on Random Number Generation and Testing, by Edward J. Dudewicz and Thomas G. Ralley. The book was published in 1981 by American Sciences Press, Inc in Columbus, Ohio.

[5] Birger Jansson (1966), Random Number Generators, Victor Pettersons Bokindustri Aktiebolag, Stockholm, pp. 26-27.

[6] Birger Jansson (1966), Random Number Generators, Victor Pettersons Bokindustri Aktiebolag, Stockholm, pp. 18,37-38.

[7] This book provides an introductory look at discrete-event simulation. It includes two chapters on random number generation and testing. It gave a better description of the Kolmogorov-Smirnov test than I was able to give. The book was written by Jerry Banks and John S. Carson, II in 1984. The title is Discrete-Event System Simulation, and it is printed by Prentice-Hall Inc. in New Jersey.

[8] Frank J. Massey, Jr., "The Kolmogorov-Smirnov Test for Goodness of Fit", The Journal of the American Statistical Association, Vol. 46, copyright 1951, pages 68-71.

Other papers that might be of interest:

IBM Technical Report
IBM Document GC20-8011-0
IBM Data Processing Techniques
Random Number Generation and Testing

Umesh V. Vazirani, Vijay V. Vazirani
Efficient and Secure Pseudo-Random Number Generation
IEEE 1984

Acknowledgements: I would like to thank Dr. Murat Tanik and Dr. Samir Soliman for their technical help. I would like to thank Betty Clay for the time and effort she put in proofreading and criticizing this article.

KOLMOGOROV-SMIRNOV CRITICAL VALUES

Sample Size (N)	Level of Significance $D_{0.10}$	$D_{0.05}$	$D_{0.01}$
1	0.950	0.975	0.995
2	0.776	0.842	0.929
3	0.642	0.708	0.828
4	0.564	0.624	0.733
5	0.510	0.565	0.669
6	0.470	0.521	0.618
7	0.438	0.486	0.577
8	0.411	0.457	0.543
9	0.388	0.432	0.514
10	0.368	0.410	0.490
11	0.352	0.391	0.468
12	0.338	0.375	0.450
13	0.325	0.361	0.433
14	0.314	0.349	0.418
15	0.304	0.338	0.404
16	0.295	0.328	0.392
17	0.286	0.318	0.381
18	0.278	0.309	0.371
19	0.272	0.301	0.363
20	0.264	0.294	0.356
25	0.24	0.27	0.32
30	0.22	0.24	0.29
35	0.21	0.23	0.27
Over 35	$\frac{1.22}{\sqrt{N}}$	$\frac{1.36}{\sqrt{N}}$	$\frac{1.63}{\sqrt{N}}$

Source: F.J. Massey, "The Kolmogorov-Smirnov Test for Goodness of Fit", *The Journal of the American Statistical Association*, Vol.46, © 1951, page 70. Adapted with the permission of the American Statistical Association.

Draco

A Shareware Compiled Language for the Amiga

by Chris Gray

Apart from the machine's fairly steep learning curve, the greatest barrier confronting the would-be Amiga programmer is probably the high cost of language compilers. C, the most popular development language, is represented in the Amiga world by the Lattice and Manx compilers, either of which will set you back a couple of hundred dollars. Modula 2, the next in popularity, now exists in three versions (TDI, Benchmark and M2Amiga), all with similarly intimidating price tags. To date, the long-held dream of a complete public domain development system for one these languages remains unrealized, though we know of at least three public domain C compilers in various stages of completion.

An alternative that has received less attention than it deserves is Chris Gray's shareware Draco language, ported by the author from his original CP/M version in 1986. Although the present version of Draco does not support floating point (a future version probably will), it is otherwise a complete procedural language that is entirely capable of handling full-size Amiga applications. As Chris explains in the first of the two articles presented here, programming in Draco is very similar to programming in C or Pascal (or, for that matter, Modula 2). However, Draco has certain unusual features (such as the ability to pass as procedure parameters arrays whose size is not known until run-time) that the other languages lack.

The article below is adapted from a section of the documentation accompanying the original Draco distribution (Fish Disks 76 and 77), and explains part of the philosophy of Draco and why it was written. We have also included a short programming example (again from the original distribution), to give you an idea of the flavour of Draco code. This program, "strings", extracts ASCII strings from a binary file. We have added a few comments to clarify some syntax details. The second article, written especially for this magazine, explores some internal technical details about the Amiga implementation of the Draco compiler.

* * * * *

I usually describe Draco (pronounced Dray-ko) as a "systems programming language". That means that it is a language that is suitable for what I think of as systems programming - writing operating systems, compilers, editors, databases, and so on. This is not to say that it isn't suitable for other applications, such as writing games, graphics programs and numerical programs; only that the language has all the facilities needed for the former type of programming, such as bit operators, pointer manipulation and support for complex data structures.

What is different about Draco? I won't try to compare it with every other programming language in the world; instead I'll stick to two of the most popular ones for micros nowadays - C and Pascal. Draco has all the facilities of C, except for bit-fields and the macro preprocessor. Unlike C, and like Pascal, it is a strongly typed language. This means that it won't let you assign an integer to a pointer (unless you really insist). It is also not an expression language like C; thus it makes a quite strict distinction between statements like "a := 27" and expressions like "a + 27". Pascal is strongly typed, but lacks many of C's facilities - pointer manipulation, bit manipulation, standard separate compilation and conditional compilation, for example. I like to think that Draco combines the best features of both languages.

Visually, Draco doesn't really resemble either language closely, but is a little closer to Pascal than C. It uses ':=' for assignment and '=' for comparison, like Pascal and unlike C. Its structure and union declarations are like those of C, however. As a simple comparison, here follows the same program, written in Pascal, C, and Draco:

Pascal:

```
PROGRAM test(INPUT, OUTPUT);
VAR
  i, j : INTEGER;

BEGIN
  FOR i := 0 TO 10 DO BEGIN
     FOR j := 0 TO i DO
        WRITE(j : 2, ' ');
     WRITELN
  END;
  WRITELN("All done.")
END.
```

C:

```
#include <stdio.h>

main(argc, argv)
int argc;
char *argv[];
{
    int i, j;

    for (i = 0; i <= 10; ++i) {
        for (j = 0; j <= i; ++j)
            printf("%2d ", j);
        printf("\n");
    }
    printf("All done.\n");
}
```

Draco:

```
proc main()void:
    int i, j;

    for i from 0 upto 10 do
        for j from 0 upto i do
            write(j : 2, ' ');
        od;
        writeln();
    od;
    writeln("All done.");
corp;
```

First, it's clear that the C program has lots of brackets, while the Pascal and Draco programs have lots of keywords. A significant difference, not very clear in this small example, is that Draco uses different keywords for each job, rather than relying on a single construct (the BEGIN/END or left brace/right brace block). I greatly prefer keywords, finding them easier on the eye. I also prefer languages in which the use of case (upper vs. lower) is available for my own purposes, not equivalent as in most Pascals. Also, note that the Draco program uses 'upto' in the 'for' loops - this tells the compiler that the loop will be counting upwards; 'downto' is used for downward counting loops. C doesn't really have a semantically different 'for' loop - it's just a kind of shorthand for a 'while' loop.

Some of the inadequacies of Pascal from my point of view are:

- no standard separate compilation
- no conditional compilation
- no general string mechanism
- no pointer manipulation
- no bit manipulation
- I *hate* BEGIN and END!
- no signed/unsigned types
- limitations on function and argument types
- procedure calls don't use '()' - they look like variables
- no typed, named, constants
- no available, decent implementations (fast compilation, good code, nice libraries, good error reporting)
- I/O semantics are poor for interactive programs
- no file inclusion or module specification facility

Some of the inadequacies of C from my point of view:

- too many bloody brackets!
- horrible declaration syntax (just what is "char *(*p[])()"?)
- error prone conventions (how many times have *you* written '=' when you meant '=='?)
- non-portable I/O (if you don't believe this, take a look at the open calls on CP/M or MS-DOS versions of C, where you get to tell it what it's supposed to do with '\n')
- potential for very unreadable code (misuse of macros, etc.)
- slow compilers (as I've heard it, the reason the original Unix C compiler for the PDP-11 generated assembler source was so that the compiler writers didn't have to worry about long/short branch optimization, since that was done by the assembler. Producing assembler source is just plain slow. Those who argue that they want to hand edit it to improve it are crazy!)
- lack of much type checking (I prefer compilers that tell me about my dumb mistakes. This is a lot better in the ANSI draft version.)
- inefficient standard setup - passing everything as 16 bits on an 8 bit CPU isn't so hot (or 32 bits on a 16 bit one)
- stupid linkers - why add all that code I'm never calling?
- no built-in I/O - this makes even the simplest programs large - no typed, named constants

All of these issues have been addressed in the Draco language and tools. Just as important to me is the quality of the tools (compiler, linker, etc.) The Draco compiler goes from source code to relocatable, optimized machine code at a rate of about 2000 lines per minute on a 5 MHz 8080. Working from one 8" floppy disk, the entire compiler (about 10,000 lines) can be rebuilt in under 10 minutes. No other compiler I've heard of for CP/M can do this (at least not and produce good code). The linker will link small programs in one quick pass, and won't load any code that isn't referenced by the program. A simple "hello there, world" program is under 1000 bytes. The Amiga version of the compiler operates at about the same speed, and the linker I'm using (BLink) isn't bad, but doesn't selectively load like mine does.

So I've written my very own personal compiler, that does things just the way I want; why should anyone else want to use it? Put simply, the Draco package (which includes the various libraries I've built up) is possibly the most effective way to produce compact, efficient code for CP/M systems. In the past couple of years, aside from fine tuning the compiler, I've written somewhere around 20,000 lines of Draco code, including the screen editor I'm typing this into, a complex graphic role-playing game, several CRT-oriented games for CP/M, ranging from the trivial to the quite complex, a database package, a text processor (with a friend), a modem program, and others. If you want to program a CP/M system, whether for

fun, profit or whatever, and are willing to learn another language, then I feel that Draco is a valid choice. The Amiga version hasn't been used as much yet, but appears to be reasonably solid. I plan on bringing it up to the standard I set with the CP/M-80 version, but this will take time and a lot of work.

To be fair, I will end this intro with a list of things that I find are lacking in this version of Draco:

- essentially non-existent floating point support
- no proper modules (although Draco goes about half way to providing a usable kind of module)
- no bit oriented type (I haven't yet fully convinced myself that this is needed)
- error handling is considerably better than most C compilers I've heard of, but it could still use some improvement
- object code can *always* use improvement, but the improvements that are left would either be difficult or of little actual benefit, and would probably make the compiler too big to fit on standard CP/M systems. This isn't as much of a problem on the Amiga, but I do plan on keeping the compiler small enough to run well on a 512K machine.

and, of course,

- Draco is supported only by me, and available only on the systems that I choose to put it on (currently CP/M-80 and Commodore Amiga).

Q U A L I T Y

8 megabytes for the Amiga!

ASDG designs and builds in quality from the start. Our 8 meg board comes with 0, 2, 4, 6, or 8 meg installed. Of course it's no-wait-state memory, of course it's fully auto-configuring, and comes in your choice of the Amiga® 1000 or Amiga® 2000 form factor. You wouldn't expect less from ASDG.

We back up our quality with a full 18 month warranty. With memory boards from ½ meg to 8 meg ASDG gives you more choices for memory expansion. For more information contact ASDG at
(608) 273-6585
925 Stewart Street
Madison, Wisconsin 53713

ASDG Inc.
(608) 273-6585

P E R F O R M A N C E

"strings.d": A Draco Programming Example

```
#include:util.g

int                                 /* constant declarations      */
   DEFAULTMINLENGTH = 4,
   MAXLENGTH = 255;

proc main() void:
   channel input binary chin;       /* local variables            */
   file() fyle;
   *char par;
   byte b;
   ushort minLength, length, i;
   bool inString;
   [MAXLENGTH] char buffer;

   par := GetPar();                 /* read next command line arg */
   if par = nil then                /* par = nil if no more args  */
      writeln("Use is: strings [-n] file1 ... fileN");
      exit(1);
   fi;
   minLength := DEFAULTMINLENGTH;
   if par* = '-' and (par + 1)* >= '0' and (par + 1)* <= '9' then
      minLength := (par + 1)* - '0';
      par := GetPar();
   fi;
   if par = nil then
      writeln("Use is: strings [-n] file1 ... fileN");
      exit(1);
   fi;
   while par ~= nil do
      if not open(chin, fyle, par) then
         writeln(par, ": does not exist.\n");
      else
         inString := false;
         while read(chin; b) do
            if b >= 0x20 and b <= 0x7e then
               if not inString then
                  inString := true;
                  length := 0;
               fi;
               if length = MAXLENGTH - 1 then
                  for i from 0 upto MAXLENGTH - 20 do
                     write(buffer[i]);
                  od;
                  for i from MAXLENGTH - 19 upto MAXLENGTH - 2 do
                     buffer[i - (MAXLENGTH - 19)] := buffer[i];
                  od;
                  length := 18;
               fi;
               buffer[length] := b + '\e';   /* \e is a null byte */
               length := length + 1;
            else
               if inString then
                  if length >= minLength then
                     for i from 0 upto length - 1 do
                        write(buffer[i]);
                     od;
                     writeln();
                  fi;
                  inString := false;
               fi;
            fi;
         od;
         close(chin);
      fi;
      par := GetPar();
   od;
corp;
```

Draco

The Language and the Compiler

by Chris Gray

In this article I will attempt to outline some of the somewhat non-standard features of the Draco language, and describe some of the innards of the compiler and why they are the way they are. To do that, I will start with some history, since it greatly affected Draco. That will be followed by some of the decisions I made and some of the consequences of them. Following that I will get into some details on such matters as code generation and declaration representation. The section on code generation won't mean much to you unless you are generally familiar with the 68000 instruction set.

Some History

Back when I was first thinking about writing a compiler for my new microcomputer (an Exidy Sorcerer with a Z-80 CPU running CP/M-80), one compiler I was familiar with was the C compiler running under Unix version 6 on a PDP-11 computer. Here is a sketch of the compilation process under that compiler (given a source file "test.c" in my current directory that I want to compile into fast-running code):

I begin by typing:

```
cc -O test.c
```

Unix finds and loads the program "cc", a small executive program that oversees the running of the several programs that make up the actual C compiler. The "cc" program notices the arguments, notes that I want my program optimized, and that I have a C source file. It calls up a program ("cpp", the C preprocessor) to handle the reading of #include files, macro expansion and conditional compilation. The output of the preprocessor is another C source file, which is fed into the third program ("c0"), which parses and translates the program into an intermediate form. This intermediate form goes into the fourth program ("c1"), which translates it into PDP-11 assembler source code, which is written to a temporary file. Because I wanted my program optimized, "cc" now calls up the fifth program ("copt" - the optimizer), which reads the assembler source, does some machine-specific optimizations on it, and writes it back out to another temporary file. Now "cc" calls the sixth program ("as" - the assembler), which reads the source file twice during its operation, and produces an output file called "test.o". Finally, "cc" calls the seventh and last program, ("ld" - the linker), which reads "test.o" twice while it is linking it with various libraries of support routines, finally producing a runnable program in file "a.out".

As you can guess, the whole operation took quite a while. I thought about implementing such a system on my single 256K floppy disk, and decided it was out of the question. What I wanted was a compiler that would read the source file once, and directly produce an unlinked object file without needing any intermediate file storage. I then wanted a linker that would read the object file once, link it with any needed libraries, and write the resulting executable program. Both programs were to run on a system that gave me about 56K of memory to play with. Also, the compiler was to produce good object code, and it had to do so quickly. These goals weren't present during the early stages of the project, but materialized as it became clear that what I wanted was indeed possible. An additional requirement, that the compiler/language do considerably more checking of the program than typical C compilers do, came about from my experience with C and my experience with an earlier compiler I had written for the IBM 360/370 that did little checking.

Some Consequences and Decisions

The above requirements led me to come up with some basic requirements of the language and compiler (some of which may have been incorrect):

- the compiler had to be strictly one-pass. No part of the compilation process must need information from a later point in the input file.
- the compiler had to operate on a procedure-by-procedure basis. After a procedure had been compiled, no information about it (other than its declaration) could be preserved.
- the compiler had to directly produce object code. I couldn't rely on an assembler phase for anything.
- the compiler couldn't do dynamic memory allocation - the memory overhead would be too severe for the constraints I had.
- the text of error messages would have to be in a file, which was to be accessed by the compiler. This is acceptable, speed-

wise, since most compilations during program development have few if any errors (not counting logic errors of course!)

Some other aspects came about from later decisions:

• I wanted as few language/compiler restrictions as possible. Identifiers could be as long as you had room for. Big procedures must be compilable. Array dimensionality should be limited only by memory. (Just for fun, I have tried this out on my 2.5 Meg Amiga - the compiler's fixed size descriptor table wouldn't let me nest 'for' statements more than 18 deep, so I only used an 18-dimensional array, but if that's too limiting for anyone, I can build them a compiler with a bigger table!)
• all identifiers (with one small exception) must be declared before they are used.
• the compiler has only one symbol table, which contains all user identifiers as well as the language's reserved words. This was one reason why the language doesn't let you have multiple meanings for a given identifier at any single point in a program.
• the two largest memory consumers - the object code buffer and the symbol text buffer, are in one large area and grow towards each other. An artifact of this is that identifiers are stored backwards. If you scan the object code of the compiler, you will see the backwards text of all of the reserved words and predefined symbols.
• numeric types are 'normally' unsigned - signed types are 'unusual'. This is the reverse from C, where signed is 'normal'. This partially came about because of the lower efficiency of signed operations on the 8080 family of CPUs. I still believe it was the proper choice, however, though there are some consequences that the programmer should be aware of:

In C, if you multiply a signed int and an unsigned int, you will be doing an unsigned multiply, since the presence of the 'abnormal' unsigned value forces the 'abnormal' mode. In Draco, the operation would be signed. This is the same for all operators, but the results of addition, subtraction, and the bitwise operators are not affected. Draco does not allow comparisons between signed and unsigned values other than the equality comparisons (= and ~=), since this can be a source of very subtle bugs.

Watch out for 'for' statements. One of the form:

```
for i from 1 upto count do
    . . .
od;
```

is harmless, but one of the form:

```
for i from 0 upto count - 1 do
    . . .
od;
```

will not work if 'count' is zero and unsigned, since subtracting one from it will yield an underflow, and the largest value of its type.

Draco does not allow the shift operators to be used on signed values. This is a fairly recent restriction, but is added to avoid the problem C has with different CPUs doing different things on signed shifts.

• the compiler uses recursive descent techniques to parse all aspects of the program. This tends to allow better error messages than the table-driven approach often used in C compilers. To illustrate this, recall that boolean 'and' has higher precedence than boolean 'or', then think about the following fragment from the compiler:

```
proc parseBoolAnd()void:

    parseBoolNot();
    while Token = T_AND do
        /* type-checking, code generation, etc. */
        scan();                /* skip the 'and' */
        parseBoolNot();
        /* type-checking, code generation, etc. */
    od;
corp;

proc parseBoolOr()void:

    parseBoolAnd();
    while Token = T_OR do
        /* type-checking, code generation, etc. */
        scan();
        parseBoolAnd();
        /* type-checking, code generation, etc. */
    od;
corp;
```

Code Generation

With no dynamic allocation and no temporary files, the compiler can't build much in the way of data structures describing the program. Thus, the code generation is intermixed with the parsing, so that what information there is can be used.

To conserve code space, the routines used in code generation should be as general as possible, so that they can be used in many places. This is aided by the relative uniformity of the addressing modes on the processors (none, in the case of the 8080!). For example, the 68000 version of the compiler has a routine called 'getMode', with the following introductory comment and declaration:

```
/*
* getMode - given a DescTable entry, generate auxiliary
*           code as needed and return mode/register/extra
*           information needed to address it.
*/

proc getMode(*DESCRIPTOR d; *byte pMode;
             *ushort pReg; *uint pWord;
             *bool pFreeA, pFreeD)void:
```

Most values are passed out of 'getMode' through pointer parameters. The DESCRIPTOR value contains such information as whether the value is a constant, a local variable, a global variable or a temporary in a register; its value if a constant or its offset in the group if not; and its type (so we know whether it will fit in a register or not). The mode returned is the 68000 addressing mode used to access the value (data register direct, address register direct, immediate, address register indirect with offset, and so on). The 'pReg' and 'pWord' parameters return any register number involved and any offset from that register. Parameters 'pFreeA' and 'pFreeD' indicate whether or not the caller is to free an address and/or a data register after using them to access the value. 'getMode' is basically a large case statement that does different things for different types of values. It knows about limitations in the 68000, such as the fact that offsets are +/-32767 unless there is an index register, in which case they are +/-127, and can get around these by generating intermediate instructions (to explicitly add in an offset that is too big, for example). The next pair of routines ('tailStuff' and 'opTail') use 'getMode' to actually generate the more uniform instructions produced by the compiler. Here's the first bit of 'opTail':

```
/*
* opTail -generate the instruction portion based on the value in
*         the Top-of-Stack descriptor. Only instructions with
*         an "effective address" operand are generated here.
*/

proc opTail(OPTYPE opType; uint opCode; byte leftMode;
            ushort leftReg; bool extraAddr, extraData)void:
   byte ea;
   bool freeA, freeD;

   getMode(&DescTable[0], &TailMode, &TailReg,
            &TailOffsetWord, &freeA, &freeD);
   ea := TailMode << 3 | TailReg;
```

It is given an opcode, the class of the opcode, and some information about the left-hand operand for this operator. It calls 'getMode' to find out how to address the right-hand operand, which is described by a descriptor at the top of 'DescTable', a stack of value descriptors. After ensuring that all needed registers are up-to-date (they may have been stacked by intervening code in some cases), it does a case statement based on the type of the opcode, and calls mode-specific code generation routines that simply do bitwise shifts and or's to construct the final instruction. 'opTail' is in turn called by 'modedBinary', which looks at the current two operands, and can completely generate code to perform an operation on them, so long as the 68000 has that instruction in the general form. This includes add, subtract, bitwise 'and' and bitwise 'or'. It in turn is called directly from the various recursive descent routines.

As a final bit of this gory detail, the handler of the assignment statement is able to make use of the 68000's nice general move instructions. Here's the code that does most of the assignment statements in Draco:

```
   getMode(&DescTable[1], &lMode, &lReg, &lWrd,
            &lFreeA, &lFreeD);
   getMode(&DescTable[0], &rMode, &rReg, &rWrd,
            &rFreeA, &rFreeD);
   needRegs((lFreeA - false) + (rFreeA - false),
            (lFreeD - false) + (rFreeD - false)
   );
   opMove(
      if size = S_BYTE then
         OP_MOVEB
      elif size = S_WORD then
         OP_MOVEW
      else
         OP_MOVEL
      fi,
      rMode << 3 | rReg,
      lMode << 3 | lReg
   );
   tailStuff(&DescTable[0],  true, rMode, rReg, rWrd, rFreeA,
            rFreeD);
   tailStuff(&DescTable[1], false, lMode, lReg, lWrd, lFreeA,
            lFreeD);
```

To summarize all of this, there is a chain of routines, specific to the target CPU, which are called from the recursive descent parsing routines, and which generate code for the language constructs being parsed. They all emit code into a large buffer, which is written out at the end of the current procedure. (The code must be buffered so that forward branches can be patched up when the compiler knows where they are branching to.)

There are both advantages and disadvantages to this scheme of code generation. Advantages:

- requires less code
- all handling of a given construct is together
- information for special-casing is directly available
- requires no memory for intermediate data structures

Disadvantages:

- retargetting the compiler to another CPU requires changes to the parsing routines
- no opportunites for global optimizations
- individual routines are more complex
- lacks 'modularity'

Symbol Table and Types

As mentioned previously, the Draco compiler maintains only one symbol table. This too has its consequences. It is slightly harder to have multiple meanings for a given symbol (though a C compiler I wrote at work also uses only a single symbol table - the various definitions are linked together in a stack, so that the latest one is deleted when its defining scope is exited). The advantages of a single symbol table are realized when it is the lexical scanner that is responsible for inserting and looking up symbols. Then, no symbol need ever be copied, and noth-

ing is ever looked up more than once (if the symbol isn't found, the lexical scanner inserts it into the table, marking it as undefined). All users of a symbol are simply given a pointer to its entry in the symbol table. Declaration of the symbol fills in the definition; use of it retrieves the definition (or complains if the symbol is marked as undefined). Putting the reserved words into the symbol table works when the language treats them as reserved words - the user cannot ever declare symbols with those names. (Some languages, such as PL/I, treat things like 'IF' and 'THEN' as keywords - the user can declare variables with those names, and the words are special only in positions where their special meaning makes sense.)

Types in Draco can be arbitrarily complex (another one of my requirements). Thus we can write things like:

```
[2, 2]*[3] struct {int a; enum {a1,a2} b; [3][4] char c} thing;
```

and the compiler (if not the reader of the program!) will be perfectly happy. This means that the type representation scheme used by many C compilers (which limits the number of levels of '*', '[]' or '()') will not work. It appears that a tree structure is needed. This is represented in Draco by a pair of arrays. The first is an array of type descriptors, which are of a fixed, quite small, size. The second is an array of extra type information, from which we will allocate various-sized chunks to hold structures needed for the more complex types. When a type needs this extra information, a field in the basic type descriptor contains an index into the extra information table of the start of the structure, and the compiler resorts to type cheating (using a union type) to reference the information. A type itself is always represented by an index into the type table. In the 8080 version of the compiler, the number of types was kept less than 256, so that a one-byte value could be used as a type. For the Amiga version, I wanted to make sure that *all* of the Amiga includes could be handled in a single source file, so the tables are considerably larger, and types are 16 bit values. The various types requiring extra information are:

- arrays - number of dimensions, and size in each dimension
- structures/unions - total size, number of fields, and pointer to symbol table entry for each field
- procedure types - result types, parameter count, types of parameters
- operator types - bit vector of operations, base type, base name

Languages vary greatly in the rules they use for type equivalence. In Draco, the rule can be summarized as "if two types *look* the same, they *are* the same, unless one or both have been given a name". The language description on the distribution disks goes into this in more detail. As far as the compiler is concerned, the key is the representation of a named type, and the way the compiler adds types to its type table. A named type is passed around as a type table entry that simply points at another entry describing the type that was named. This indirection entry is skipped over when operations on the type are required, but not when type checking of the type itself is done. When any type is added to the type table, the table is scanned for another occurrence of the type; if one is found it is used and the new one thrown out. Thus two types that look the same are in fact the same type. The check simply does not count named type entries as equivalent to anything else. If two named types name equivalent types, they will both point to the same type, but will have separate named-type entries to distinguish them.

Optimization

Code optimization, or more properly, code improvement, is one of the fun parts of compiler writing. There are many schemes and techniques that can be used to improve the code produced by a compiler. Most of the more esoteric ones (such as inlining, code motion, common subexpression elimination and some forms of strength reduction) cannot be done in a one-pass compiler like Draco - they require information about whole ranges of the program. (Actually, I've got some ideas about the last three based on having all of the initial object code available in the code buffer.) The 8080 version of the compiler does the following types of optimizations:

Code optimization is one of the fun parts of compiler writing. There are many schemes and techniques that can be used to improve the code...

- special casing: such things as replacing multiplies by adds or shifts where possible; using special instructions on the CPU, such as the 68000's MOVEQ, ADDQ; etc.
- peephole optimization: looking at short sequences of code and replacing them with equivalent, but better, short sequences of code. A common one here that is applicable to the 68000 is the move-move optimization, which gets rid of redundant move instructions.
- state remembering: this can be a variety of things, but essentially remembers what is in the machine's registers so that if they are needed again, they are readily available. A common one here is the store-load optimization, which gets rid of loads of variables into registers when the variable has just been stored from that register.

The 68000 version of the Draco compiler currently only has the first type. Adding the other two types could result in considerably improved code, as could adding register variables (the latter shouldn't be too difficult, since the register handling was taken from one of my 360/370 compilers, which did have register variables). One major problem with the current 68000 compiler that isn't covered by any of the above is that it wants to save registers used in a function at the beginning of the function, but it doesn't know which registers will be used until it has compiled the whole function. Backpatching the MOVEM instruction used isn't sufficient, since that changes

the offsets needed to access parameters, which are on the other side of the variable number of saved registers. This problem results in a high overhead for procedure calls, and unnecessary consumption of stack space.

General Code Characteristics

Because of its fairly full use of the 68000's addressing modes, the code produced by the Draco compiler can appear to be locally optimal. This can be quite misleading, however, since a CPU with as many resources as the 68000 requires more global considerations to generate really good code. Draco's code should be fairly recognizable to someone trying to debug it at the assembler level, but you must watch out for Draco's fairly full use of compile-time operations. As an example, the following source:

```
proc test()void:
    type
        inner_t = struct {
            enum {red, green, blue} colour;
            [100] char buffer;
        },
        test_t = struct {
            int field1;
            [10, 10] inner_t field2;
        };
    test_t t;
    *test_t pt;

    t.field2[3,7].buffer[99] := pt*.field2[0,0].colour-red+'A';
corp;
```

produces the following object code:

```
1. MOVEM.LD2-D7/A2-A6,-(A7)
2. SUB.W#277A,A7
3. MOVE.L(A7),A6
4. MOVE.B0002(A6),D7
5. ADD.B#41,D7
6. MOVE.BD7,0F03(A7)
7. ADD.W#277A,A7
8. MOVEM.L(A7)+,D2-D7/A2-A6
9. RTS
```

The first two lines are the procedure entry code - they save all of the registers and allocate space on the stack for the local variables. Similarly, the last three lines are the procedure exit code - they deallocate the space for the locals, restore the registers and return to the caller. Line 3 loads the value of local variable 'pt' into address register 6, so that it can be dereferenced. Note that DRACO does *not* use a frame pointer; all references to local variables and parameters are done through the stack pointer, A7; the compiler keeps track of additional offsets introduced by the stacking of temporary values and parameters. Special casing in the compiler has noticed that the required offset is 0, so that register indirect addressing can be used.

Line 4 fetches the single byte enumeration value 'pt*.field2[0, 0].colour' into the lower byte of data register 7. The compiler has done the constant array indexing and field offset calculations at compile time, resulting in a total offset of 2 from the pointer in A6. Line 5 adds the character 'A' (hex 41) to the byte value in D7. The subtraction of 'red' does not appear because the value of 'red' is 0 (special casing again). Finally, line 6 stores the resulting character value into 't.field2[3, 7].buffer[99]'. Again, all constant array indexing and field offset calculations have been done at compile time, yielding an offset of hex f03, which is within the 68000's range for the address-register-indirect-with-displacement addressing mode.

Using the 'code' Construct

I have heard many people claim that the only proper way for a compiler to operate is to produce assembler source for the program. About the only point to this that I can see is that it lets people readily see what code the compiler has generated. The immense disadvantage is that the compilation process is significantly slowed down. For a compiler like Draco, which produces machine code directly, you simply use a disassembler to see the code (one is available on an early Fish disk).

I have heard of people who claim that you want the assembler source so that you can go in and tweak it to get a bit more speed out of it. That is a maintenance nightmare - any change to the original source will require re-editing of the intermediate assembler code. I personally would never be quite sure that any automated editing was doing it correctly.

A second reason for wanting the compiler to produce assembler source is so that things not do-able by the compiler can be done with in-line assembler code. For those few cases where doing the whole thing in an assembler subroutine is not appropriate, Draco has the 'code' construct, which allows arbitrary instructions to be inserted into the generated code, albeit not quite as prettily as in-line assembler.

The one use of this that I've made using the Amiga version of Draco is in writing a handler to be inserted into the input stream (see the description of the 'input.device' in Volume 1 of the Amiga Rom Kernel Manual). Here is an abstract of that routine:

```
/*
* am_inputHandler -
*           Magic routine that is called by Exec when the
*           Input Device has something for me.
*/

proc am_inputHandler()*InputEvent_t:
    uint
        OP_MOVEL = 0x2000,
        M_ADIR = 0o1,
        M_DISP = 0o5,
        R_SP = 7,
        A0 = 0;
```

```
    *InputEvent_t ie, head;

    /* move A0 into 'head' */
    code(
        OP_MOVEL | R_SP << 9 |
            M_DISP << 6 | M_ADIR << 3 | A0,
        head
    );
    /* 'head' now contains the head of the linked list of input
        events. We scan through them, doing any processing as
        desired, and removing those that we don't want to
        pass through to Intuition. */
    ie := head;
    Forbid();
    while ie ~= nil do
        /* ... */
        ie := ie*.ie_NextEvent;
    od;
    Permit();
    head
corp;
```

Be *very* careful when using the 'code' construct. It knows nothing about what 68000 instructions look like, so it will readily accept garbage. A particularly nasty bug-causer is the fact that the type of a constant expression determines how big it will be when stuck into the code. In the above example, the first 16 bits of the MOVE instruction must be of type 'uint' or 'int', so that it is inserted as a 16 bit value. The type of a constant expression in Draco is the largest of the types of its component parts. If, for example, I had used the constant '0' instead of 'A0' above, I would have generated a 32 bit (and wrong) initial part of the instruction, since constants default to type 'ulong'.

Using the name of a local variable directly in a 'code' construct yields the 16 bit offset of that variable from the top of the stack, *assuming that nothing else has been stacked.* A global variable generates a 32 bit relocatable value that will be relocated to the variable's address. Any other value must be a constant expression, and will generate an 8, 16 or 32 bit value depending on the type of the constant expression. As in the above example, the 68000 instruction encodings are generated at compile time by the appropriate shifting and or'ing of values.

The Future

It's clear that lots of things can be done to improve the Amiga Draco compiler. Will they get done, and why haven't they already been done? The answer to the first question is "probably", and the answer to the second is "because I've had too much fun doing other things". Since getting Draco ported, I've been spending my time on a number of projects:

- Amiga-izing my CRT library
- adding multiple windows to Ded (the Draco editor)
- various small projects, just playing around with the Amiga
- writing Amiga Empire (Amiga Empire is a totally new implementation of the Peter Langston Empire, which is a multiplayer game of exploration, economics, war, etc. lasting a couple of months. It is nearly twenty thousand lines of code, so it took a fair amount of work (as does the continuing playtesting!)
- Amiga-izing my 'Explore' system. This is a graphical/text D & D style adventure system. It now runs on the Amiga, with the scenario called 'Amelon'. Much of the conversion work was with the 'ExpCre' program, used to graphically create the non-text parts of scenarios (maps, mazes, images, palettes, etc.). I could release this project now, but it doesn't do justice to the Amiga. I'd like to add smooth-scrolling maps, filled-in mazewalks, loadable pictures, and maybe even sound. This is what I would like to work on next.

Like most people, however, my plans are subject to change. If there is a lot of interest in Draco, and demand for some of the improvements, I could be persuaded to work on it instead of Explore. A month of good work on Draco could make some significant improvements. Here are some of the things that can be done:

- fix entry/exit to not save all the registers
- get 'operator types' working
- add floating point support
- add a peep-hole optimizer
- add 'state-remembering' optimizations
- add register variables
- port my own linker, and switch the compiler back to using my object-module format
- clean up error handling, code generation, etc.

One of the reasons I haven't been tempted to work on the current compiler is that I have some fairly grandiose plans for the next version. It is to be an integrated system...

One of the reasons I haven't been tempted to work on the current compiler is that I have some fairly grandiose plans for the next version. It is to be an integrated system including a truly optimizing code generator, a better Draco front end, a programmable editor, a programmable command shell, a programmable database package, a syntax-checking editor, a game based on programming, perhaps even a spreadsheet tied in with all the rest. The Amiga's sharable libraries would be at the heart of it. I have already written a lexical-scanner library (which needs floating point support added), but there is enough work in the whole project to keep me busy for a couple of years.

So folks, what do you think? Should I take the time to upgrade the existing compiler (which has proved to be essentially bug-free), should I get Explore out, or should I go for the new compiler system?

Representing Objects in a CAD System

A Data Structure To Simplify CAD Program Design

by Charles B. Blish

Charles Blish is president and founder of SoftCircuits Inc., a Florida-based Amiga software development company. SoftCircuits' products for the Amiga include the PCLO and PCLO-plus electronic CAD packages discussed here; Scheme, a program to assist in generating electronic schematic drawings; and the driver software shipped with the C Ltd. hard disk drives.

The purpose of this article is to introduce readers to a high-performance alternate method of maintaining a CAD system database inside a CAD program. It is my hope that this information will be of use to others in the creation of other types of specialized CAD systems besides printed circuit board (PCB) systems.

Typically, CAD systems are general purpose: they use a few types of structures that can be sized and placed independently. Examples of these traditional items are lines, circles, arcs and points. Often, these are *all* that are used in a CAD system. You create more complex items by creating groups of the simpler items at particular sizes, and in specific relative positions. To draw an area of a design that is made up of these items, one must first determine which of them lie within, or partially within, the segment of the design that is going to be displayed. Then each item that does not lie entirely upon the display area must be 'clipped', or cut, where it 'falls off' the screen. This is very time and computation intensive.

Long waits or other inconveniences imposed by the program directly affect the quality of the work. Performance is the central issue...

This method of representation, while simple, has other major drawbacks besides display computation time. While it is easy enough to maintain a database of these simple items, a complex object may require a large number of them to be represented accurately. Immediate consequences are a large, variable memory load to represent an object, sequences of drawing operations directly related to the complexity of the object, and display update times that vary according to the data being displayed. There are also effects upon how the program can manipulate the data - these are not of immediate concern to the program's user, though they certainly concern the programmer who must provide the CAD tool with a method to manipulate these items.

What follows is a method that removes all of these drawbacks, in exchange for some simple compromises. The particular implementation in this case is a printed circuit board program, an application where circles, lines, and arbitrarily shaped areas are all used. This is not a theoretical implementation - this is the specific method that is used in PCLO, a product that has been in use on the Amiga since April of 1986.

We believe that in a design process like that of PCB generation, the CAD program's responsiveness - its overall speed - is of primary importance. Since design of a PCB is a very complex process, one that demands a great deal from the PCB designer, any long waits or other inconveniences imposed by the program are more than just irritating - they directly affect the quality of the work. For this reason, performance is the central issue.

In the following discussion, the term *item* refers to an entity such as a trace or a pad on the PCB, while the term *object* will be used to specify a building block that is used alone or in combination with others to create one of these items.

Rather than creating items from primitive lines and circles, we wish to represent them by joining a small set of objects that have the characteristics of that area of the item we wish to use the object to create.

This sounds complex, but is not. Simply put, we are going to select a set of objects that, taken together in different ways, can represent the final item. Think of them as pieces of a puzzle. Each piece is a square that can be placed next to any other square in the database.

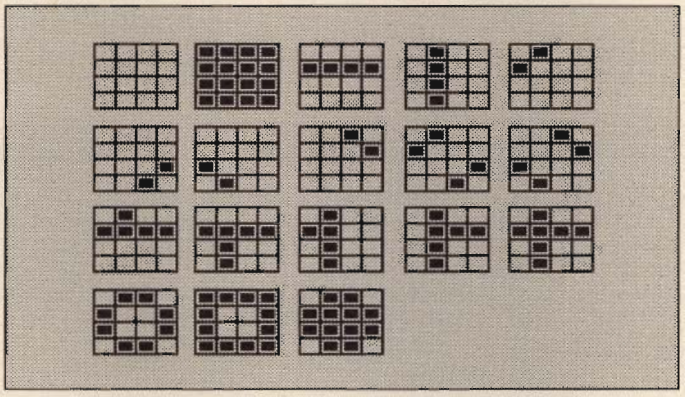
Instead of using a variable amount of memory for each item, we will create a workspace that consists of a two dimensional

array of cells of a particular size, each containing one of these objects at all times. Initially, each cell will contain an object that represents the "empty cell" condition.

In order to accomplish this, and have it be useful, we must make certain that the objects meet the following criteria:

1) They must be *very* easy to draw.
2) They must be memory-efficient (one byte is ideal).
3) There must be a small, limited set of them.
4) They must be able to describe any item we wish to create.
5) They must be objects that can be manipulated easily.

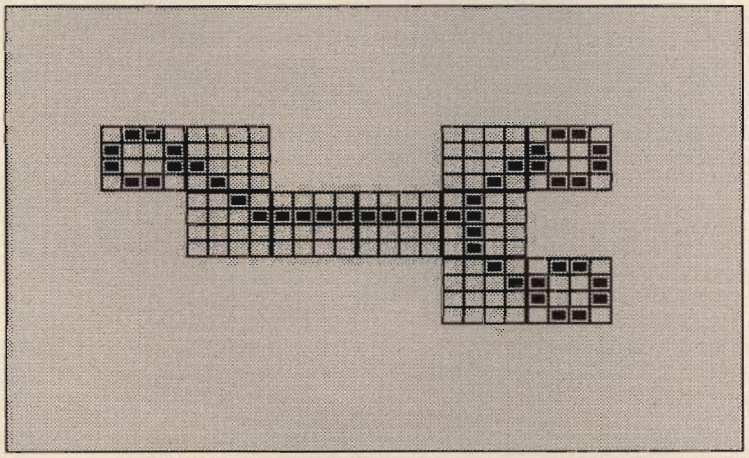
Examining the requirements of a printed circuit board led us to create the set of 18 objects depicted in the diagram at right:



The set of building blocks, or 'objects', that form the basis of PCLO's data handling. These can be flexibly combined to create higher-level 'items' (see figure below).

These meet requirement number one very well. We use an area of the screen 4 pixels across by 4 pixels down to represent one of these objects. We only draw objects if they would appear completely on the screen, so the routine that draws them only has to decide where to put them, and whether to put them in the right or left nybble of the target bytes on the screen. A screen redraw consists of examining all of the locations that are within the window area, and drawing the proper image for that area into the screen. Given a 320 by 200 screen, the job is completed before you can see it happen, no matter *what* data is contained in the area that is represented by the display.

At right is an example of how these objects are used in combination to create actual useful PCB items. This diagram represents three pads connected by a single trace.



The representation of an 'item' in PCLO. Although not every conceivable item can be represented using the available 'objects', the performance gain futweighs the loss of complete generality.

Requirements 2 and 3 are also satisfied - there are less than 256 objects, so they can be represented in one byte. We now choose the workspace size based on this. Since, in this type of CAD representation, every area must be defined, we must be conservative. Two hundred and fifty six locations along each side of the work area provides enough room for most PCB applications - 12.8 inches. This will use 65535 bytes for one layer of data. Since most PCB designs require two layers, that makes 128k bytes in total. This kind of memory usage is not a problem for the Amiga.

Using the assumption that there are always two layers we can use to represent the objects, you can actually represent this set of objects in 4 bits, saving quite a bit of memory - 64k. How? If we assume pads are objects that exist on both layers, then the value in the cell for the top layer can specify one of the 15 single surface object types, or a general pad-type indicator. The corresponding cell in the second layer can specify the type of pad - solid, with a hole, square, etc., if the first layer contains the pad indicator, or else is interpreted as a single surface object type. We choose not to do this; the reason is simply performance. It is much easier, and therefore faster, to process a cell that completely describes itself without referring to another. In addition, looking up cells that are packed into nybbles is more time consuming.

The fourth requirement is that we can create any item we need to. Here, we have done well, although for some small percentage of PCB designs, this method, as outlined here, will not be sufficient.

SoftCircuits also produces a more sophisticated PCB CAD program, called PCLOplus. It uses the same methods, but instead of a cell that defines a .050 inch square in a layer, it uses one that defines a square .025 inches on a side. This allows the program to define the workspace to a precision four times greater than PCLO. Specific structures that can be created within this environment are things like two conductors being routed between adjacent pads on an integrated circuit, called 'fine-line' PCB artwork in the industry.

Using this method, we use the objects, or combinations of them, to describe that the item exists; its precision and its placement are a function of the final drawing or plotting routines only. We know that each item exists in a

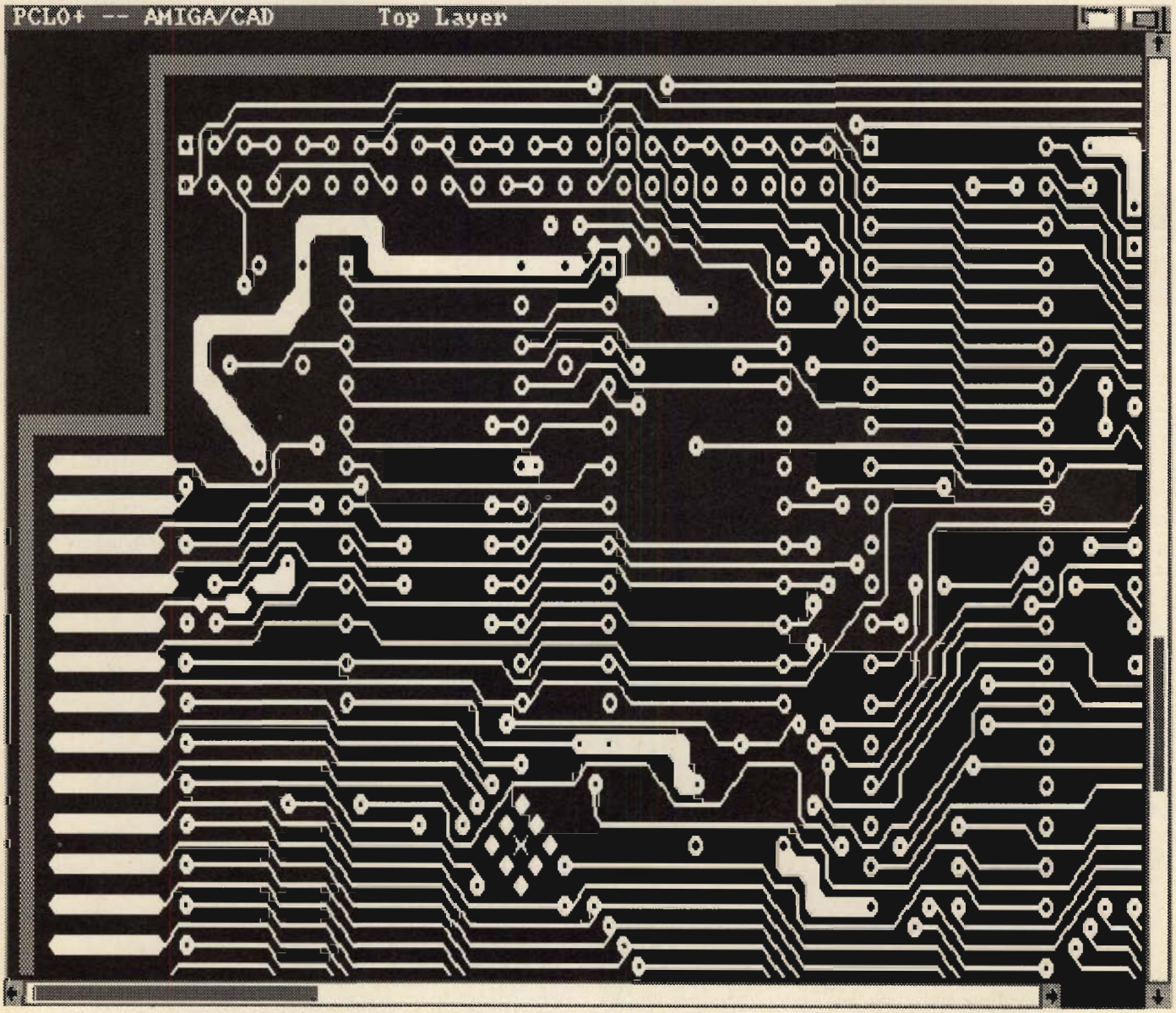
cell that is .050 inches square; it is up to the final routines to render the artwork in such a manner that the PCB is precisely represented in a way that will be produceable and functional. This includes centering the traces represented by the codes in the cells, trimming areas of solid conductor so there are no sharp corners in the final artwork, filling 'inside' sharp corners, and other similar tasks.

As far as item five - manipulating the objects - is concerned, methods exist or can be created for doing almost any type of transformation or analysis on a PCB item made up of these objects. PCLO implements routing, ripup, rotation, move, copy, and many more. Actually, trace manipulation is *easier* when the PCB is defined as a grid of cells like the one we are discussing.

Some simple rules are used when defining how a particular operation's scope is determined in PCLO. First, trace segment object types are 'connected' when they are of types that cause pixels to be immediately adjacent horizontally, vertically, or diagonally. Secondly, solid area objects and pad-type objects are only connected when they are adjacent to each other horizontally and vertically. Solid areas that are adjacent to each other diagonally do *not* connect, and are trimmed at final drawing time so as to provide adequate clearance to guarantee that no connection will occur.

You could use this method to describe plumbing, hydraulics, or any system that can be represented in a two-dimensional area, given only that it can be described by a limited set of objects you could 'build' larger items out of.

I hope that this information has helped you to understand CAD representation and implementation better, and that it may be of some use to you in deciding how to structure another CAD system.



IFF picture of a printed circuit board layout created with PCLOplus, a CAD program from SoftCircuits Inc. The colours in the original 640x400 picture have been remapped as follows: yellow to white, red to light grey, green to dark grey, black to black.

Programming With AmigaDOS

Update - An Intelligent File Copier

by Bob Rakosky

Bob Rakosky is a senior consultant with AGS Information Services, Inc., and is the author of saf-T-net, a commercial hard-disk backup utility for the Amiga.

In this article, I will be demonstrating techniques for programming in the AmigaDOS environment and for utilizing the ARP library, by presenting a file copying utility that I developed called Update. The program is written in 68000 assembly language, but could easily be translated to C, Modula 2, or any other programming language that allows access to the Amiga Operating System library calls. I chose to write this in assembly because I wanted to keep the size of the executable module small, so that it could easily fit on most Workbench disks.

Update - the program

When developing a large program, most people will have many different source code modules that get compiled and linked together to form the executable program. When I'm working on such a project, I will keep all of my program files in a single directory on my work disk, and do the coding, compiling, and debugging there. I also keep a separate diskette with (almost) current copies of my program files.

The problem arises when it's time for me to take a 'checkpoint', and ensure that this spare set of files is current. For a smaller project, I can just recopy all files from my development area, but with a large project this can be very time-consuming.

That's where Update comes in. Update copies a set of files into a directory, just like the CLI 'Copy' command. Unlike Copy, however, Update will copy a file only if it is newer than the file with the same name in the target directory. In an alternate mode, Update copies all files except those that are older than a particular file specified by the user. The input files can be specified as a list of file names and/or directory names, and can include wild-card patterns.

Unlike Copy, Update will copy a file only if it is newer than the file with the same name in the target directory...

Update is a CLI-only command and is invoked from the CLI prompt with a command of the format:

Update [FROM] obj1 [obj2 ... obj12] TO target [SINCE file]

where obj1...obj12 are the names of files and/or directories (and may contain wild card patterns) specifying all input files to be considered for copying. Note that directories are *not* processed recursively, so that directories contained within specified directories are not processed. The target specified in the command line (TO option) is the name of the directory to receive the copied files. If the SINCE option is specified, all input files whose datestamp is greater than or equal to that of the 'since' file will be copied. If this option is not specified, the datestamp of each input file is compared with the like-named file in the output directory, and if the input file has a later datestamp (or if the corresponding output file does not exist) it will be copied.

Like most of the AmigaDOS CLI commands, Update also will prompt you for the arguments when invoked with the command:

Update ?

You will then be presented with a standard DOS-style argument template, which in this case will look like:

FROM/A,,,,,,,,,,TO/K/A,SINCE/K

and you will be asked for your desired options. The ARP command replacements have introduced an extension to this standard, which Update also provides (since it uses the ARP command-line parsing routine). Like all of the standard ARP commands, responding to this prompt with another question-mark (?) will present you with more detailed help.

What is ARP?

ARP is an acronym for "AmigaDOS Replacement Programs". It consists of a number of programs written to replace the stan-

dard CLI utility commands supplied with the Amiga, such as assign, dir, type, list, etc. These commands are written mostly in assembler, and are designed to eliminate the need for the BCPL environment in which the CLI normally resides. A more concrete benefit to users is that the ARP commands tend to be smaller in size than their BCPL counterparts, and support the full functionality of the programs they replace in addition to a few more 'bells and whistles'. ARP is available as a freely distributable package, developed primarily by Charlie Heath (of Microsmith's TxED fame) and Scott Ballantyne (whose article on ARP appeared in the previous issue of *Transactor for the Amiga).*

Besides the supplied command replacement programs, the ARP package includes a disk-based run-time library in standard Exec format. This library contains many common routines used by the ARP programs, and must be placed in the LIBS: logical directory of your system in order for the ARP commands to function. Documentation is provided for using the library functions in your own programs (for examples see the accompanying source code to Update).

Using the ARP routines provided me with a number of advantages. First, the use of the ARP command-line processor makes it easier for me, the programmer, to provide a consistent command-line interface to the user (via the '?' for the command template and subsequent '?' for more detailed help routines). Secondly, using the ARP library routines really saved me a lot of time in the development of Update; in fact, if it weren't for ARP, I most likely would not have made the deadline for this article. Thirdly, the use of the ARP routines just made the development cycle that much easier, since I didn't have to develop unique command-line parsing routines or the complex wild-card matching routines utilized by Update.

The ARP routines provided me with a number of advantages... If it weren't for ARP, I most likely would not have made the deadline for this article...

Update does not take advantage of the ARP routines in all cases. Certain routines were coded utilizing direct AmigaDOS calls, where in fact the ARP library provides similar capability. This was done intentionally in order to illustrate the use of the AmigaDOS calls. In other cases, I decided to bypass ARP because many of the ARP resource allocation and tracking routines that I could have used are designed for the allocation of resources to be automatically freed at program exit time (actually when the CloseLibrary routine is called to disconnect from the ARP library). My use of resources within Update differs from this philosophy, in that many of the resources I allocate (such as files opened) should be freed long before Update is finished with its processing. Consequently, I use the standard DOS and Exec functions, being careful to ensure that I always free what I allocate, either when I am done with it or at the end of the program, depending on the needs for that particular resource. The ARP resource trackers didn't seem to fit my needs for that level of flexibility.

How Does Update Work?

The command line is parsed using the ARP 'GADS' library routine. This routine is passed the command line data and length, a command line template (in standard AmigaDOS format such as one creates for an execute file), a string of text that will be presented if the user types a '?' in response to the arguments prompt, and an array of longwords into which the routine will place the results of the parse. The routine will return a boolean value indicating whether the command line conforms to the requirements as specified in the command template. If the parse is successful, the longword array will be filled in with the values (pointers) supplied for each parameter specified in the template, in the exact order in which they appear in the template.

After the command line is parsed, the 'since' file is validated, if one is specified, and the datestamp for that file is obtained for subsequent comparisons. The specified output directory is then checked to ensure that it does exist and that it is a directory, not a file.

The program now processes each of the (up to 12) input objects. This iterative process forms the body of the program, which is performed by the routine labelled 'procarg' (refer to the accompanying source code listing). Since each input object may be either a file or a directory, and can include wild-cards as part of its name, I have used the ARP FindFirst and FindNext routines to search the file system and construct a list of (zero or more) files that match the specified argument. This is a very useful function pair in that it takes care of a major job that I did not have to code: it examines the directory implicitly (in relation to the current directory) or explicitly (when the full path is specified in the argument) and finds all files that match the argument, including wild-card pattern matching. This routine is not really necessary if the argument does not contain wild-cards, but will work for this type of argument also, returning the same type of information as would be obtained by performing a Lock() and an Examine() for the object.

Each object located in the loop is tested to determine whether it is an actual data file, or a directory. In the case of a data file, a routine is performed to decide whether to copy it or not (the routine labelled 'dofile'). In the case of a directory, each file contained within the directory is considered for processing. This process is not recursive: sub-directories are bypassed.

In processing an individual file (code section 'dofile'), a subroutine called 'getcdate' is called to obtain a datestamp for comparison. If the SINCE option was specified, the datestamp of the 'since' file has already been obtained. If this option was not specified, a file name is constructed consisting of the target directory name, with the input file name appended. Note

that the input file name is taken from the FileInfoBlock structure, which ensures that only the file name itself (with no directory or path information) is appended. This file is located (by obtaining a shared lock) and, if it exists, the datestamp is extracted for comparison.

If the datestamp of the input file is greater than the comparison date, the input file is copied to the target directory. Once the data is copied, the datestamp of the output dataset is altered to match that of the input file.

Working With AmigaDOS

With the proper documentation, working with the AmigaDOS library routines is not too difficult. (Bantam's *The AmigaDOS Manual*, among other sources, provides the necessary level of documentation.) The major things to watch out for are the apparent inconsistencies between BCPL-oriented data items, and C-oriented data items. Some of the DOS routines want BCPL-type pointers and strings passed to them or returned by them, and others work with C-type pointers and strings. Check the documentation carefully, and if something doesn't seem to be working as documented, look for this type of confusion. A good debugger is an extremely useful tool.

BCPL deals with an address as a long-word number, so the 'address value' of 4 (for example) refers to the 4th long-word in the memory space (relative to zero), which is at memory location 16 ($10 in hex). To convert a BCPL-type pointer to a C-type pointer, simply multiply by 4 (or shift the value 2 bit positions to the left). To convert a C pointer to a BCPL pointer (BPTR), simply divide the address by 4 (or shift right by two positions). You will note that unless the C-type address is divisible by 4, the conversion to a BPTR will not be correct, since BCPL always assumes that data are long-word aligned. This is why many of the DOS routines (such as Examine) require that the data area passed to them be long-word aligned. When allocating storage for accessing DOS routines, it is necessary to keep these alignment requirements in mind.

When coding in a high-level language, you don't always have control over the alignment of data areas allocated in your program (most will ensure word alignment if needed, but not necessarily long-word alignment). The easiest way around this is to use the Exec library's AllocMem routine to allocate the memory you need. AllocMem always allocates memory starting at a long-word boundary, so it is guaranteed safe for use with AmigaDOS. When coding at the assembler level, the "C-NOP 0,4" assembler directive requests that the assembler align the next instruction/data definition statement at a long-word boundary. This is the technique I used in Update.

The other major difference between BCPL and C-type variables is the format of character strings. With C-type strings, the data is stored in consecutive locations, with a null byte (binary zero) indicating the end of the string data. With BCPL, string information is stored with the first byte containing a binary number (up to 255) that indicates how many bytes of character data follow. Also, as noted above, BCPL expects a pointer to a string to be a BPTR. There is a section of code in Update's 'setdate' routine that builds a BCPL-type string (or 'BSTR') from a standard C-type string. (There are also ARP routines that convert between BCPL-strings and C-strings, but they didn't quite fit my needs in this case, and it doesn't require much code to do it inline.)

AmigaDOS Structures

When using AmigaDOS functions for file manipulations, there are three major control blocks that you will be concerned with: the FileLock, the FileInfoBlock, and the FileHandle. You seldom need to be concerned with the contents of either the FileLock or the FileHandle, as they mainly contain information useful for the library routines themselves in the internal processing of your requests. They are obtained for you by AmigaDOS in response to certain types of requests, and you just need to keep track of them. The fact that each is actually a BCPL pointer to a data structure is transparent to most program's requirements.

*A lock is like a licence obtained from AmigaDOS to make use of a file. Locks you obtain **must** be freed before the program exits...*

AmigaDOS uses locks (as described by Lock structures) to keep track of which files are being used. A lock is like a licence obtained by a program from AmigaDOS to make use of a file. Any lock obtained by your program *must* be freed before the program exits, or the file corresponding to that lock will be considered 'in use' until you reboot your machine, possibly interfering with the activities of other programs wanting access to the same file.

There are two types of locks. Exclusive locks (ACCESS_WRITE) can only be obtained if no other program is using the file and, once obtained, will prevent any other program from accessing the file until you have finished with it and returned the lock to the system. Shared locks (ACCESS_READ) may be obtained (if the file exists) as long as no program currently owns an exclusive lock for the file, and will allow many programs to access the file simultaneously. Attempting to obtain a shared lock on a file is a good way to test if the file exists, since it incurs much less overhead than attempting to open the file. Locks are returned by a number of AmigaDOS library calls, but the most direct way of obtaining one is to use the AmigaDOS Lock function call which, when passed the name of a file (as a standard C-type string) and the type of lock required, will return the lock (in the form of the BCPL address of a Lock structure) to your program.

The FileInfoBlock is a control block used by AmigaDOS to pass information about a file back to your program via the Examine and ExNext routines. Your program needs to allocate the storage for this FileInfoBlock, making sure that the storage

area is longword aligned. When you pass the address of this storage area (as a normal pointer, this time, not a BPTR), along with a lock for the file, to the Examine AmigaDOS library function, the information about that file will be filled into your FileInfoBlock. Useful information contained in the FileInfoBlock includes the file's name, last-modified-date/time, file protection status bits, file comment and file size. Also included is an item ('fib_DirEntryType') that contains an indicator as to whether the locked object is a data file or a directory. The value of this field will be greater than 0 if the object is a directory, negative if it is a file. By iteratively calling the ExNext function, you can get information for all of the files and sub-directories contained in a directory. This is the standard means of walking the directory chains in the AmigaDOS file system.

The FileHandle is the third of the AmigaDOS file-manipulation structures used by programs. It is obtained by opening a file using the AmigaDOS Open function, and is passed back to DOS whenever some action (reading or writing) is to be performed on the contents of the file. A FileHandle is 'returned' to AmigaDOS by calling the Close function. The internal contents of the FileHandle structure are generally of no direct interest to a program, but are used by AmigaDOS itself to keep track of file-positioning information.

Programming Style

Even those of you who are not fluent in 68000 assembler can benefit from reading the source code file accompanying this article. The program is written in a modular format, and each subroutine section is commented at the beginning with a block comment that briefly describes the purpose of the section, the arguments passed to it, and the information that is returned by the routine. It is also annotated in a way that should be meaningful for C programmers (or programmers in other languages who have learned to deal with C-oriented Amiga documentation). All system (and ARP) library calls are preceded by comments that show the C-style syntax of the library calls, as well as the documented register usage conventions for passing the arguments directly from Assembler. The format is exactly that used in the standard documentation from Commodore-Amiga.

When you call an operating system library function (like the AmigaDOS functions) from a high-level language like C, your use of a function name (such as Open or Examine) in most cases invokes not the function itself but a 'binding stub' that calls the function on your behalf. The reason for this arrangement is that the library functions expect to receive parameters in CPU registers, whereas most high-level languages *pass* parameters on a stack in memory. The binding stub (or 'glue routine') takes care of moving the parameters off the stack and into the registers where the OS function will be looking for them.

In assembler, you have immediate control of the CPU registers, so this indirect approach is not necessary. Instead, you code the call as a JSR instruction (Jump to SubRoutine) to an offset from the library's base address (provided via the OpenLibrary Exec function call). The offset used for a given routine is defined as an external reference (resolved at link time by scanning the link-type library - usually "amiga.lib"), and is named _LVOroutine-name.

The source code for Update hides this fact by using a macro named CALL defined at the beginning of the file. Thus, I code an invocation of the Open function with the source statement "CALL Open(a6)", which actually generates the assembler statement "JSR _LVOOpen(a6)". Note that register a6 must have been first loaded with the base address of "dos.library" in order for this statement to work.

Writing Reentrant Code

One other point of interest about the Update program: the program is totally "reentrant", which means that multiple tasks can execute the same copy of the program code at the same time without conflict. While the standard CLI environment will provide for multiple copies of a program to be loaded into memory if multiple tasks invoke the same program simultaneously, MetaComco has included a 'Resident' command as part of their Shell environment, which allows a single copy of the program to be used by multiple tasks. Many programs will not function correctly (or at all) if used in this manner, but Update will.

The Update program is totally reentrant - multiple tasks can execute the same copy of the code at the same time without conflict....

In order for a program to be reentrant, it must not modify any part of the code or data sections of the program itself. Notice that all modified data items used by Update are in the section of the program for which memory is allocated upon entry to the program, and are addressed as offsets to the pointer to that memory (maintained in register a5). Each simultaneous execution of Update will have allocated different segments of memory to use as variable space. Coding reentrant programs with high-level languages is much more difficult, because most of the compilers place any static variables into data and bss hunks in the executable file, which means that simultaneous tasks sharing the program code would also be sharing the same copy of these data areas.

Summary

I hope that by reading this article and examining the accompanying source you have obtained an understanding of some of the techniques used for manipulating files using the standard AmigaDOS library functions. Also, I hope I have illustrated some of the techniques needed to use routines provided in the ARP.library, which greatly simplifies the development of Amiga programs.

```
*****************************************************************
*                                                               *
*   Program Name: Update                                        *
*        Version: 1.0                                           *
* Change History: 11-Nov-87 Initial Program Version             *
*                                                               *
*  Copyright (c) 1987 - Bob Rakosky                             *
*                                                               *
*  USAGE:                                                       *
*   Update [FROM] obj1 [obj2 ... obj12] TO dir [SINCE mfile]    *
*                                                               *
*   where obj1 ... obj12 are either files or directories,       *
*             (with or without wildcards)                       *
*         dir is the output directory name                      *
*         mfile is a file name                                  *
*                                                               *
* Will copy all files specified (or in directories specified)   *
* to dir IF the file's datestamp is later than the datestamp    *
* for file mfile (if specified) or later than the datestamp     *
* of the corresponding-named file in dir2 (if mfile not         *
* specified). In the latter case, if the corresponding file     *
* does not exist in the output directory, the file is copied.   *
*                                                               *
*  NOTES                                                        *
*      This program REQUIRES the use of ARP.library, which      *
*      must be in your libs: logical directory. If the          *
*      library is not present, the program will issue an error  *
*      message and exit.                                        *
*                                                               *
*****************************************************************
        include "exec/types.i"
        include "exec/memory.i"
        include "libraries/dos.i"
        include "libraries/dosextens.i"
        include "arpbase.i"
*---------------------------------------------------------------*
*                  Some commonly used macros                    *
*---------------------------------------------------------------*

XTERN   MACRO           ;XTERN <reference>
        XREF    _LVO\1  ; Adds the '_LVO' prefix to it.
        ENDM

CALL    MACRO           ;call < routine(register) >
        jsr     _LVO\1  ; Adds the '_LVO' prefix to the routine name.
        ENDM

*---------------------------------------------------------------*
*                            Equates                            *
*---------------------------------------------------------------*
Absbase EQU     4

*---------------------------------------------------------------*
*                      External references                      *
*---------------------------------------------------------------*
        XTERN   OpenLibrary
        XTERN   CloseLibrary
        XTERN   AllocMem
        XTERN   FreeMem
        XTERN   Output
        XTERN   Write
        XTERN   Read
        XTERN   Lock
        XTERN   UnLock
        XTERN   ParentDir
        XTERN   Examine
        XTERN   ExNext
        XTERN   Open
        XTERN   Close
        XTERN   IoErr
        XTERN   DeviceProc
*---------------------------------------------------------------*
*      Buffer Size for Read/Write Buffer                        *
*---------------------------------------------------------------*
RW_BUFS EQU     $4000           ;16 K Buffer size

*---------------------------------------------------------------*
*                      Work Area Offsets                        *
* Note: the following are offsets into an area obtained using   *
* AllocMem. The locations are addressed relative to register a5 *
*---------------------------------------------------------------*

cmdline EQU   0                 ;pointer to command line
cmdlen  EQU   4                 ;length of command line
dosaddr EQU   8                 ;address of DOSBase
arpbase EQU  12                 ;address of ArpBase
stdout  EQU  16                 ;file handle of std output
databuf EQU  20                 ;pointer to 16K data buffer for copying files
argv    EQU  24                 ;argument vectors for input names (FROM)
todir   EQU  72                 ;arg vector for TO keyword
sinfil  EQU  76                 ;arg vector for SINCE file
cmpdate EQU  80                 ;DateStamp for compare
work    EQU  92                 ;work word (for printf)
outfnam EQU  96                 ;pointer into outbuf for file name
pktargs EQU 100                 ;arg array for DOS Packets
pktarg0 EQU pktargs
pktarg1 EQU pktargs+4
pktarg2 EQU pktargs+8
pktarg3 EQU pktargs+12
argc    EQU 128

fib     EQU 132
anchor  EQU fib+fib_SIZEOF+(fib_SIZEOF&3)
fname   EQU anchor+AP_SIZEOF+(AP_SIZEOF&3)
outbuf  EQU fname+128
dirbuf  EQU outbuf+128
strwork EQU dirbuf+128
wk_lgth EQU strwork+128


*****************************************************************
*                           Main Program                        *
*****************************************************************

start   movem.l d1-d7/a0-a6,-(a7) ;save entry regs

        bsr     init            ;perform initialization
        tst.l   d0              ;Q. Good Return?
        bne     exit            ;A.  no - quit now

        bsr     parse           ;Parse the command line
        tst.l   d0              ;Q. Good return?
        bne     exit            ;A. No - exit

        tst.l   sinfil(a5)      ;Q. SINCE file name specified?
        beq.s   main010         ;A. no since file - bypass
        move.l  sinfil(a5),a0   ;address of SINCE file name
        bsr     getfib          ;get datestamp of file
        tst.l   d0              ;Q.  Good Return?
        beq.s   main010         ;A.  Yes - continue
*---------------------------------------------------------------*
*        Unable to locate/examine 'SINCE' file                  *
*---------------------------------------------------------------*
        lea     nosinm(pc),a0
        move.l  a0,d2           ;address message
        move.l  #nosinml,d3
        move.l  stdout(a5),d1
        move.l  dosaddr(a5),a6
*  Write(file,data,length)
*         d1   d2    d3
        CALL    Write(a6)
        moveq.l #20,d0          ;set bad return code
        bra     exit            ;and quit
```

```
main010:
        lea     fib(a5),a1          ;point to fib
        lea     fib_DateStamp(a1),a0  ;point to Datestamp
        lea     cmpdate(a5),a1      ;save 'since date' here
*-----------------------------------------------------------------*
*       The datestamp for comparison will be one tick less        *
*       than that of the 'SINCE' file.                            *
*-----------------------------------------------------------------*
        move.l  (a0)+,(a1)+         ;move datestamp days to compare area
        move.l  (a0)+,(a1)+         ;move datestamp minutes to compare area
        move.l  (a0),d0             ;get ticks
        subq.l  #1,d0               ;decrement by 1
        move.l  d0,(a1)             ;move ticks to compare area

*-----------------------------------------------------------------*
*       Verify Output specified is a directory                    *
*-----------------------------------------------------------------*
        move.l  todir(a5),a0        ;address of output directory name
        bsr     getfib              ;get file info
        tst.l   d0                  ;Q.  Good Return?
        beq.s   main020             ;A.  Yes - continue
        move.l  stdout(a5),d1       ;msg file
        lea     noout(pc),a0
        move.l  a0,d2               ;address message
        move.l  #nooutl,d3          ;length of it.
        move.l  a4,a6               ;DOSBase into a6
*   Write(file,data,length)
*          d1    d2    d3
        CALL    Write(a6)           ;Send message to standard out
        moveq.l #20,d0              ;set return code
        bra.s   exit                ;and quit

main020:
        lea     fib(a5),a0                  ;point to file info block
        move.l  fib_DirEntryType(a0),d0     ;file or directory indicator
        bpl.s   main030                     ;> 0 ==> directory - continue
*-----------------------------------------------------------------*
*       TO specified is not a directory                           *
*-----------------------------------------------------------------*
        move.l  stdout(a5),d1       ;msg file
        lea     notdir(pc),a0
        move.l  a0,d2               ;address message
        move.l  #notdirl,d3         ;length of message
        move.l  a4,a6               ;DOSBase into a6
*   Write(file,data,length)
*          d1    d2    d3
        CALL    Write(a6)           ;Send message to standard out
        moveq.l #20,d0              ;set return code
        bra.s   exit                ;and quit

main030:
*-----------------------------------------------------------------*
*       copy target directory name to buffer and mark end         *
*-----------------------------------------------------------------*
        move.l  todir(a5),a0        ;path name
        lea     outbuf(a5),a1       ;buffer area
1$:     move.b  (a0)+,(a1)+         ;next char
        bne     1$                  ;loop until end of string
        lea     -2(a1),a1           ;back up to last non-blank character
        move.b  (a1)+,d0            ;examine last character
        cmp.b   #':',d0             ;Q. Last character colon?
        beq.s   2$                  ;A. yes - don't append slash
        move.b  #'/',(a1)+          ;A. no - append a slash
2$:
        move.l  a1,outfnam(a5)      ;concat output file name here

*-----------------------------------------------------------------*
*   loop and process all arguments                                *
*-----------------------------------------------------------------*
        move.w  argc(a5),d7         ;number of arguments
        ext.l   d7
        move.l  #2,d6               ;reduce by at least 1 (for target)
                                    ;  plus 1 to make relative to 0
        tst.l   sinfil(a5)          ;Q. Was 'since' option specified?
        beq.s   main040             ;A. No - bypass adjustment
        addq.l  #1,d6               ;A. Yes - one less input arguments
main040:
        sub.l   d6,d7               ;adjust loop counter
        clr.l   d6                  ;index value into d6
        lea     argv(a5),a4         ;point to first arg
main050:
        move.l  0(a4,d6.w),a0       ;pick up next argument
        bsr     procarg             ;process input argument
        tst.l   d0                  ;Q.  Good return?
        bne.s   exit                ;A.  No - quit now
        addq.w  #4,d6               ;increment index
        dbra    d7,main050          ;loop on
*-----------------------------------------------------------------*
*   End of processing loop - set good return code and exit        *
*-----------------------------------------------------------------*
        clr.l   d0                  ;set good return code and exit
exit:   move.l  d0,-(a7)            ;save return code
        bsr     cleanup             ;free resources
        move.l  (a7)+,d0            ;restore return code
        movem.l (a7)+,d1-d7/a0-a6   ;restore other regs
        rts                         ;return to caller

        ; PAGE
*****************************************************************
*     Section Name: init                                        *
*        Function: Performs initial housekeeping                *
*  Entry Registers: a0 - command line string                    *
*                   d0 - command line length                    *
*   Exit Registers: d0 - return code (0 = success)              *
*                   a5 - work area address                      *
*****************************************************************

init:
        move.l  a0,a3               ;save command line
        move.l  d0,d3               ;  address and length
*-----------------------------------------------------------------*
*                       Open DOS Library                          *
*-----------------------------------------------------------------*
        move.l  Absbase,a6          ;find library
        lea     dosname(pc),a1      ;point to libname string
        moveq   #33,d0              ;version at least 1.2 (or higher)
*  libbase = OpenLibrary(libname,version);
*    d0                    a1      d0
        CALL    OpenLibrary(a6)     ;open dos library
        movea.l d0,a4               ;save dos pointer for later use
        beq     init_err            ;OpenLibrary failed, can't tell
                                    ;user - just exit

*-----------------------------------------------------------------*
*               Get Std Output File Handle                        *
*-----------------------------------------------------------------*
        movea.l a4,a6               ;DosBase into a6
*   stdout = Output();
*     d0
        CALL    Output(a6)          ;get file handle for screen
        move.l  d0,d4               ;save filehandle

*-----------------------------------------------------------------*
*                       Allocate work area                        *
*-----------------------------------------------------------------*
        move.l  #wk_lgth,d0         ;length in d0
        move.l  #MEMF_CLEAR,d1      ;type in d1
        move.l  Absbase,a6
*   area = AllocMem(size,flags);
*    d0              d1    d0
        CALL    AllocMem(a6)        ;call AllocMem

        movea.l d0,a5               ;address work area
        bne.s   init010             ;we now have a work area
```

```
*-----------------------------------------------------------------*
*               Allocate failed - tell user and exit              *
*-----------------------------------------------------------------*
        move.l  d4,d1           ;Std Output file handle
        beq.s   init_err        ;couldn't get screen handle - just exit

        lea     nomem(pc),a0
        move.l  a0,d2           ;address message
        move.l  #nomeml,d3      ;length of message
        move.l  a4,a6           ;DOSBase into a6
*   Write(file,data,length)
*         d1    d2    d3
        CALL    Write(a6)       ;Send message to standard out
        bra.s   init_err        ;and leave

*-----------------------------------------------------------------*
*               We now have a work area                           *
*-----------------------------------------------------------------*
init010:
        move.l  a4,dosaddr(a5)  ;save dos address
        move.l  a3,cmdline(a5)  ;address of command line
        move.l  d3,cmdlen(a5)   ;length of command line
        move.l  d4,stdout(a5)   ;save address of Std Output

*-----------------------------------------------------------------*
*               Open ARP Library                                  *
*-----------------------------------------------------------------*
        move.l  Absbase,a6      ;find library
        lea     arpname(pc),a1  ;point to libname string
        moveq   #ArpVersion,d0  ;version at least 1.2 (or higher)
*   libbase = OpenLibrary(libname,version);
*     d0                  a1        d0
        CALL    OpenLibrary(a6) ;open dos library
        move.l  d0,arpbase(a5)  ;save base pointer for later use
        bne     init020         ;OpenLibrary OK - initialization complete
*-----------------------------------------------------------------*
*               Unable to open Arp Library                        *
*-----------------------------------------------------------------*
        move.l  stdout(a5),d1   ;msg file
        lea     noarp(pc),a0
        move.l  a0,d2           ;address message
        move.l  #noarpl,d3      ;length of message
        move.l  a4,a6           ;DOSBase into a6
*   Write(file,data,length);
*         d1    d2    d3
        CALL    Write(a6)       ;Send message to standard out
        bra.s   init_err        ;and leave

*-----------------------------------------------------------------*
*               Initialization Complete                           *
*-----------------------------------------------------------------*
init020:
        clr.l   d0              ;set good return code
        rts                     ;  return to sender

*-----------------------------------------------------------------*
*               Initialization Error                              *
*-----------------------------------------------------------------*
init_err:
        moveq.l #20,d0          ;set error return code
        rts

        ; PAGE
*****************************************************************
*     Section Name: parse                                       *
*       Function: Parse the command line                        *
*       Entry Data: cmdline - address of command line           *
*                   cmdlen - length of command line             *
*                        NOTE - relative to ZERO                *
*        Exit Data: d0 - return code (0 = success, 20 = error)  *
*                   argc - number of parameters specified       *
*                   argv - a table of pointers to the 'words'   *
*                                                               *
*****************************************************************
parse:
        movea.l cmdline(a5),a0  ;point to command line
        move.l  cmdlen(a5),d0   ;length of command line
        lea     cmdpr(pc),a1    ;help string
        lea     argv(a5),a2     ;arg array
        lea     templ(pc),a3    ;command template
        move.l  arpbase(a5),a6  ;arpbase
*  argc = GADS(cmdline,cmdlength,help,&argv[0],template);
*   d0          a0       d0       a1    a2        a3
        CALL    GADS(a6)        ;parse the command line
        move.w  d0,argc(a5)     ;store arg count
        ble.s   pars_err        ;==-1 -> parse error
                                ;==0 parsed ok but no arguments

*-----------------------------------------------------------------*
*               Successful parse of command line                  *
*-----------------------------------------------------------------*
        clr.l   d0              ;good return code
        rts                     ;return

*-----------------------------------------------------------------*
*  Unexpected error in parsing routine - should never happen      *
*-----------------------------------------------------------------*

pars_err:
        lea     ferrm(pc),a0
        move.l  a0,d2           ;Set up and write err msg to console
        move.l  #ferrml,d3
        move.l  stdout(a5),d1
        move.l  dosaddr(a5),a6
*   Write(file,data,length);
*         d1    d2    d3
        CALL    Write(a6)
        moveq.l #20,d0          ;set bad return code
        rts                     ;and return

        ; PAGE
*****************************************************************
*     Section Name: cleanup                                     *
*                                                               *
*       Function: Frees all system resources acquired           *
*                                                               *
*       Entry Data: a5 - Address of global work area            *
*                        (if it exists)                         *
*                                                               *
*        Exit Data: none                                        *
*                                                               *
*****************************************************************

cleanup:
        move.l  a5,d0           ;address of work area
        beq.s   cleanret        ;  none - nothing to clean up
        move.l  databuf(a5),d0  ;Q. do we have a read/write buffer?
        beq.s   cln010          ;A. no - bypass freeing it
        move.l  d0,a1           ;address of area to free
        move.l  #RW_BUFS,d0     ;size of area to free
        move.l  Absbase,a6
*  FreeMem(area,size);
*          a1    d0
        CALL    FreeMem(a6)

cln010:
        move.l  arpbase(a5),d0  ;Q. Is ARP library open?
        beq.s   cln020          ;A. no - don't close it
        move.l  d0,a1           ;address of arp base
        move.l  Absbase,a6
*  CloseLibrary(libbase);
*                d0
        CALL    CloseLibrary(a6)

cln020:
        move.l  a5,a1           ;address of work area
        move.l  #wk_lgth,d0     ;length of work area
        move.l  Absbase,a6      ;exec library
```

```
*  FreeMem(area,size);
*          a1   d0
        CALL    FreeMem(a6)     ;free work area
cleanret:
        rts                     ;return to caller

        ; PAGE
*****************************************************************
*     Section Name: procarg                                     *
*                                                               *
*        Function: Processes a single input argument from the   *
*                  command-line                                 *
*                                                               *
*      Entry Data: a0 - pointer to argument, which is either    *
*                       a file or directory (with/without       *
*                       wild-cards                              *
*                                                               *
*       Exit Data: d0 - return code (0 = success)               *
*                                                               *
*****************************************************************

procarg:
        move.l  a0,-(a7)                ;push passed argument
        move.l  #128,d0
        lea     anchor(a5),a0           ;point to f/n search anchor
        move.l  d0,AP_LENGTH(a0)        ;length of fname
        move.l  (a7)+,d0                ;pop passed argument
        move.l  arpbase(a5),a6          ;ARP base
*  rc = FindFirst(pattern,chain)
*  d0              d0      a0
        CALL    FindFirst(a6)
        tst.l   d0                      ;Q.  Good Return?
        bne.s   parg_err                ;A.  No - error exit
parg010:
        lea     anchor(a5),a1           ;point to search anchor
        lea     AP_INFO(a1),a0          ;point to internal FIB
        move.l  fib_DirEntryType(a0),d0 ;file or directory indicator
        bpl     parg020                 ;> 0 ==> directory - continue
*-------------------------------------------------------------*
*       entry is a file                                       *
*-------------------------------------------------------------*
        lea     fname(a5),a1            ;point to full path name
                                        ;note - a0 still points to fib
        bsr     dofile                  ;process input file
        tst.l   d0                      ;Q. Good return?
        bne.s   parg_err                ;A. no - quit
        bra.s   parg030                 ;continue with next in chain

parg020:
*-------------------------------------------------------------*
*       entry is a directory                                  *
*-------------------------------------------------------------*
        bsr     dodir                   ;process input directory
        tst.l   d0                      ;Q. Good return?
        bne.s   parg_err                ;A. no - quit

parg030:
        lea     anchor(a5),a0           ;search anchor
        move.l  arpbase(a5),a6
*  rc = FindNext(chain)
*  d0            a0
        CALL    FindNext(a6)
        tst.l   d0
        beq.s   parg010                 ;loop on if next entry

*-------------------------------------------------------------*
*       End of chain - we are done with this argument         *
*-------------------------------------------------------------*
        lea     anchor(a5),a0           ;point to anchor
        move.l  arpbase(a5),a6          ;ARP base
*  FreeAnchorChain(chain)
*                  a0
        CALL    FreeAnchorChain(a6)

        clr.l   d0                      ;return code
        rts

parg_err:
        moveq.l #20,d0                  ;set error return code
        rts

        ; PAGE
*****************************************************************
*     Section Name: dodir                                       *
*                                                               *
*        Function: Processes a file directory included in the   *
*                  input argument list                          *
*                                                               *
*      Entry Data: fname(a5) contains the full path of the      *
*                      directory                                *
*                                                               *
*       Exit Data: d0 - return code (0 = success)               *
*                                                               *
*  NOTE:  This routine uses the AmigaDOS Examine/ExNext         *
*      functions, since wild cards are not needed               *
*                                                               *
*****************************************************************

dodir:
        movem.l d2-d3/a3-a4,-(a7)   ;save registers
        lea     dirbuf(a5),a0       ;buffer area to build file names
        lea     fname(a5),a1        ;path name of directory
1$:     move.b  (a1)+,(a0)+         ;move directory name to buffer
        bne.s   1$                  ;stop at null

        lea     -2(a0),a0           ;point at last character
        move.b  (a0)+,d0            ;inspect last character
        cmp.b   #':',d0             ;Q. Is last character ':'?
        beq.s   ddir010             ;A. yes - don't append a /
        move.b  #'/',(a0)+          ;   no - append a / for path name
ddir010:
        move.l  a0,a4               ;save beginning address for file name
*-------------------------------------------------------------*
*       Allocate memory for local FIB                         *
*-------------------------------------------------------------*
        move.l  #fib_SIZEOF,d0      ;size of fib
        move.l  #MEMF_CLEAR,d1      ;allocation type
        move.l  Absbase,a6          ;Exec base
*  *ptr = AllocMem(size,type)
*  d0              d0   d1
        CALL    AllocMem(a6)        ;allocate fib
        tst.l   d0                  ;Q.  Allocation successful?
        bne.s   ddir020             ;A.  yes - continue
*-------------------------------------------------------------*
*       Unable to allocate FIB - gotta quit                   *
*-------------------------------------------------------------*
        move.l  stdout(a5),d1       ;Std Output file handle
        lea     nomem(pc),a0
        move.l  a0,d2               ;address message
        move.l  #nomeml,d3          ;length of message
        move.l  dosaddr(a5),a6      ;DOSBase into a6
*  Write(file,data,length)
*          d1   d2    d3
        CALL    Write(a6)           ;Send message to standard out
        bra     ddir_err            ;and leave

ddir020:
        move.l  d0,a3               ;save pointer to FIB
*-------------------------------------------------------------*
*               Obtain a lock on directory                    *
*-------------------------------------------------------------*
        lea     fname(a5),a0        ;point to directory name
        move.l  a0,d1
        move.l  #ACCESS_READ,d2     ;get a shared lock
        move.l  dosaddr(a5),a6
*  Lock = Lock(name,accessMode)
*  d0          d1      d2
        CALL    Lock(a6)
        tst.l   d0                  ;Q.  Lock obtained?
        bne.s   ddir030             ;A.  yes - continue
```

```
        lea     fname(a5),a0
        move.l  a0,work(a5)     ;build argument list
        lea     work(a5),a1     ;point to argument list
        lea     dlokerr(pc),a0  ;printf pattern
        move.l  arpbase(a5),a6
*  Printf(string,*args)
*          a0     a1
        CALL    Printf(a6)
        bra     ddir_err        ;and leave

ddir030:
        move.l  d0,d3           ;save lock
*-----------------------------------------------------------------*
*       Walk the directory chain, using Examine/ExNext            *
*-----------------------------------------------------------------*
        move.l  d3,d1           ;lock
        move.l  a3,d2           ;pointer to fib
        move.l  dosaddr(a5),a6
*  success = Examine(lock,fib)
*     d0              d1   d2
        CALL    Examine(a6)
        tst.l   d0              ;Q. Successful?
        bne.s   ddir040         ;A. yes - continue

        lea     fname(a5),a0
        move.l  a0,work(a5)     ;build argument list
        lea     work(a5),a1     ;point to argument list
        lea     dexerr(pc),a0   ;printf pattern
        move.l  arpbase(a5),a6
*  Printf(string,*args)
*          a0     a1
        CALL    Printf(a6)
        bra     ddir_err        ;and leave

ddir040:
*-----------------------------------------------------------------*
*       If entry is for a directory skip it                       *
* NOTE: The program will NOT recursively walk a nested            *
*       directory structure                                       *
*-----------------------------------------------------------------*
        move.l  fib_DirEntryType(a3),d0 ;file or directory indicator
        bpl.s   ddir050                 ; >= 0 ==> directory - bypass it
*-----------------------------------------------------------------*
*       Entry is a file - build full file name                    *
*-----------------------------------------------------------------*
        move.l  a4,a0                   ;point to file path
        lea     fib_FileName(a3),a1     ;point to file name
1$:     move.b  (a1)+,(a0)+             ;append file name to path
        bne.s   1$                      ;loop until null byte

*-----------------------------------------------------------------*
*       Now process the file                                      *
*-----------------------------------------------------------------*
        move.l  a3,a0           ;point to FIB
        lea     dirbuf(a5),a1   ;point to path name
        bsr     dofile          ;process file
        tst.l   d0              ;Q.  Success?
        beq.s   ddir050         ;A.  yes - continue
*-----------------------------------------------------------------*
*       Error - free local resources and exit                     *
*-----------------------------------------------------------------*
        move.l  a3,a1           ;address of fib
        move.l  #fib_SIZEOF,d0  ;length of fib
        move.l  Absbase,a6      ;exec library
*  FreeMem(area,size);
*          a1    d0
        CALL    FreeMem(a6)     ;free work area

        move.l  d3,d1           ;lock
        move.l  dosaddr(a5),a6
*  UnLock(lock)
*          d1
        CALL    UnLock(a6)      ;free lock
        bra.s   ddir_err

ddir050:
*-----------------------------------------------------------------*
*               Get next entry in directory                       *
*-----------------------------------------------------------------*
        move.l  d3,d1           ;lock
        move.l  a3,d2           ;pointer to FIB
        move.l  dosaddr(a5),a6
*  success = ExNext(lock,fib)
*     d0              d1   d2
        CALL    ExNext(a6)
        tst.l   d0              ;Q. Success?
        bne.s   ddir040         ;A. yes - loop on to process this entry
*-----------------------------------------------------------------*
*       Insure that error is NO_MORE_ENTRIES                      *
*-----------------------------------------------------------------*
        move.l  dosaddr(a5),a6
*  error = IoErr()
*   d0
        CALL    IoErr(a6)       ;get DOS error code
        move.l  d0,-(a7)        ;save error code on stack
*-----------------------------------------------------------------*
*               Free Resources                                    *
*-----------------------------------------------------------------*
        move.l  a3,a1           ;address of fib
        move.l  #fib_SIZEOF,d0  ;length of fib
        move.l  Absbase,a6      ;exec library
*  FreeMem(area,size);
*          a1    d0
        CALL    FreeMem(a6)     ;free work area

        move.l  d3,d1           ;lock
        move.l  dosaddr(a5),a6
*  UnLock(lock)
*          d1
        CALL    UnLock(a6)      ;free lock

        move.l  (a7)+,d0                    ;pop DOS error code
        cmp.l   #ERROR_NO_MORE_ENTRIES,d0   ;Q. Expected 'error' value?
        beq.s   ddir060                     ;A. yes - ok to return
        move.l  d0,work(a5)                 ;   no - put error code in arg list
        lea     work(a5),a1                 ;point to arg list
        lea     baddir(pc),a0               ;Printf pattern
        move.l  arpbase(a5),a6
*  Printf(string,*args)
*          a0     a1
        CALL    Printf(a6)      ;print error message
        bra.s   ddir_err        ;take error exit

ddir060:
        movem.l (a7)+,d2-d3/a3-a4   ;restore registers
        clr.l   d0                  ;good return code
        rts

ddir_err:
        movem.l (a7)+,d2-d3/a3-a4   ;restore registers
        moveq.l #20,d0              ;error return code
        rts

        ; PAGE
*****************************************************************
*     Section Name: dofile                                      *
*                                                               *
*        Function: Processes a file included in the input      *
*                  argument list                                *
*                                                               *
*       Entry Data: a0 - FIB for file                           *
*                   a1 - full path name for file                *
*                                                               *
*        Exit Data: d0 - return code (0 = success)              *
*                                                               *
*****************************************************************

dofile:
        movem.l d2-d5/a2-a4,-(a7)   ;save registers
        move.l  a0,a3               ;save FIB pointer
        move.l  a1,a4               ;save name pointer
```

```
*-----------------------------------------------------------------*
*       Construct full path name for output file                  *
*-----------------------------------------------------------------*
        lea     fib_FileName(a0),a1 ;point to file name of input
        move.l  outfnam(a5),a0      ;point into buffer containing output path
1$:     move.b  (a1)+,(a0)+         ;append file name to path
        bne.s   1$                  ;loop until null-byte

*-----------------------------------------------------------------*
*               Get date for comparison                           *
*-----------------------------------------------------------------*
        bsr     getcdate        ;get datestamp for compare
        tst.l   d0              ;Q.  Good return?
        bne     dofl_err        ;A.  No - quit now

        lea     cmpdate(a5),a0          ;point to compare date stamp
        lea     fib_DateStamp(a3),a1    ;point to FIB date stamp
        moveq.l #2,d0                   ;loop count - 3 long words
dofl010:
        move.l  (a0)+,d1        ;next long word from compare date
        cmp.l   (a1)+,d1        ;compare to next long word from FIB
        blt.s   dofl020         ;branch if FIB datestamp is higher
        bgt     dofl090         ;if FIB datestamp lower - do not process
        dbra    d0,dofl010      ;check entire DateStamp
        bra     dofl090         ;FIB date is equal to - skip file

dofl020:
*-----------------------------------------------------------------*
*  File is to be processed - tell user                            *
*-----------------------------------------------------------------*
        move.l  a4,work(a5)     ;build argument list
        lea     work(a5),a1     ;point to argument list
        lea     fline(pc),a0    ;printf pattern
        move.l  arpbase(a5),a6
*  Printf(string,*args)
*          a0      a1
        CALL    Printf(a6)
*-----------------------------------------------------------------*
*       Open Input file                                           *
*-----------------------------------------------------------------*
        move.l  a4,d1            ;pointer to file name
        move.l  #MODE_OLDFILE,d2 ;access mode
        move.l  dosaddr(a5),a6
*  file = Open(name,accessMode)
*   d0          d1      d2
        CALL    Open(a6)        ;Open input file
        tst.l   d0              ;Q. open successful?
        bne.s   dofl030         ;A. yes - continue
*-----------------------------------------------------------------*
*       Input file failed to open - inform user and exit          *
*-----------------------------------------------------------------*
        lea     iopnerm(pc),a0
        move.l  a0,d2           ;Set up and write err msg to console
        move.l  #iopnerl,d3
        move.l  stdout(a5),d1
        move.l  dosaddr(a5),a6
*   Write(file,data,size)
*         d1    d2    d3
        CALL    Write(a6)
        bra     dofl_err

dofl030:
        move.l  d0,d4           ;save input file handle
*-----------------------------------------------------------------*
*               Open output file                                  *
*-----------------------------------------------------------------*
        lea     outbuf(a5),a0    ;point to output file path/name
        move.l  a0,d1            ;pointer to file name
        move.l  #MODE_NEWFILE,d2 ;access mode
        move.l  dosaddr(a5),a6
*  file = Open(name,accessMode)
*   d0          d1      d2
        CALL    Open(a6)        ;Open output file
        tst.l   d0              ;Q. open successful?
        bne.s   dofl040         ;A. yes - continue
*-----------------------------------------------------------------*
*       Output file failed to open - inform user and exit         *
*-----------------------------------------------------------------*
        lea     oopnerm(pc),a0
        move.l  a0,d2           ;Set up and write err msg to console
        move.l  #oopnerl,d3
        move.l  stdout(a5),d1
        move.l  dosaddr(a5),a6
*   Write(file,data,size)
*         d1    d2    d3
        CALL    Write(a6)
        move.l  d4,d1           ;Close input file
        move.l  dosaddr(a5),a6
*   Close(file)
*         d1
        CALL    Close(a6)
        bra.s   dofl_err

dofl040:
        move.l  d0,d5           ;save output file handle
        move.l  d4,d1           ;input file handle in d1
        bsr     copyfile        ;copy the files
        move.l  d0,-(a7)        ;save copy return code

*-----------------------------------------------------------------*
*       Close the files                                           *
*-----------------------------------------------------------------*
        move.l  d5,d1           ;output file handle
        move.l  dosaddr(a5),a6
*  Close(file)
*        d1
        CALL    Close(a6)       ;close output file
        move.l  d4,d1           ;input file handle
        move.l  dosaddr(a5),a6
*  Close(file)
*        d1
        CALL    Close(a6)       ;close input file

        move.l  (a7)+,d0        ;restore return code from copy
        tst.l   d0              ;Q. Good return from copy?
        bne.s   dofl_err        ;A. no - quit
*-----------------------------------------------------------------*
*       Set DateStamp of output file to match input               *
*-----------------------------------------------------------------*
        move.l  a3,a0           ;FIB of input file
        bsr     setdate         ;call set date routine

        lea     foklin(pc),a0
        move.l  a0,d2           ;Set up and write ok msg to console
        move.l  #foklinl,d3
        move.l  stdout(a5),d1
        move.l  dosaddr(a5),a6
*   Write(file,data,length)
*         d1    d2    d3
        CALL    Write(a6)

dofl090:
        clr.l   d0
        movem.l (a7)+,d2-d5/a2-a4   ;restore registers
        rts

dofl_err:
        moveq.l #20,d0              ;set error return code
        movem.l (a7)+,d2-d5/a2-a4   ;restore registers
        rts

        ; PAGE
*****************************************************************
*     Section Name: getcdate                                    *
*                                                               *
*        Function: Obtains the datestamp of the corresponding   *
*                  output file (if since option not used)       *
*                                                               *
*       Entry Data: outbuf(a5) contains full path name of       *
```

```
*                    output file                          *
*                                                         *
*       Exit Data: cmpdate(a5) will contain the date for  *
*                    comparison                           *
*                d0 - return code (0 = success)           *
*                                                         *
*      NOTE:  if 'since' option was specified, this routine *
*             will just return.  The cmpdate field has    *
*             already been populated with the correct date *
*                                                         *
***********************************************************

getcdate:
        tst.l   sinfil(a5)      ;Q. SINCE file specified?
        beq.s   gcdt010         ;A. No - proceed
        clr.l   d0              ;A. yes - set good return code
        rts                     ;       and return
gcdt010:
        movem.l d2-d4,-(a7)     ;save work registers
*-----------------------------------------------------------*
* Obtain a lock on the output file - if file does not exist *
* this will fail.  In this case, set cmpdate to zeros so the *
* comparison will result in the file being processed        *
*-----------------------------------------------------------*
        lea     outbuf(a5),a0   ;point to output file path/name
        move.l  a0,d1
        move.l  #ACCESS_READ,d2 ;get a shared lock
        move.l  dosaddr(a5),a6
*  Lock = Lock(name,accessMode)
*   d0          d1      d2
        CALL    Lock(a6)
        tst.l   d0              ;Q.  Lock obtained?
        bne.s   gcdt020         ;A.  yes - continue
        lea     cmpdate(a5),a0  ;A.  no - point to compare date
        moveq.l #2,d0           ;loop control - set 3 longwords to 0
1$:     clr.l   (a0)+           ;set datestamp to 0's
        dbra    d0,1$           ;   for 3 long words
        movem.l (a7)+,d2-d4     ;restore registers
        clr.l   d0              ;set good return code
        rts                     ;and return

gcdt020:
*-----------------------------------------------------------*
*             Now get File Info for file                    *
*-----------------------------------------------------------*
        move.l  d0,d4           ;save Lock
        move.l  d0,d1           ;lock in d1
        lea     fib(a5),a0      ;point to work fib
        move.l  a0,d2           ;FIB pointer in d2
        move.l  dosaddr(a5),a6
*    success = Examine(lock,FIB)
*      d0              d1   d2
        CALL    Examine(a6)
        move.l  d0,-(a7)        ;save Examine return code on stack
*-----------------------------------------------------------*
*       Free the lock obtained above                        *
*-----------------------------------------------------------*
        move.l  d4,d1           ;lock into d1
        move.l  dosaddr(a5),a6
*   UnLock(lock)
*          d1
        CALL    UnLock(a6)      ;free the lock

        move.l  (a7)+,d0        ;restore Examine return code from stack
        tst.l   d0              ;Q.  Success?
        beq     gcdt_err        ;A.  No - examine failed

        lea     fib(a5),a0          ;point to fib
        lea     fib_DateStamp(a0),a1 ;point to file's datestamp
        lea     cmpdate(a5),a0      ;point to comparison datestamp
        moveq.l #2,d0               ;loop control - move 3 long words
1$:     move.l  (a1)+,(a0)+         ;move file's datestamp to compare field
        dbra    d0,1$               ;move all 3 long words
        movem.l (a7)+,d2-d4     ;restore registers
        clr.l   d0              ;good return code
        rts                     ;return to caller

gcdt_err:
        lea     exerm(pc),a0
        move.l  a0,d2           ;set up and write err msg to console
        move.l  #exerl,d3
        move.l  stdout(a5),d1
        move.l  dosaddr(a5),a6
*   Write(file,data,length)
*          d1   d2    d3
        CALL    Write(a6)
        movem.l (a7)+,d2-d4     ;restore registers
        moveq.l #20,d0          ;set bad return code
        rts                     ;and return

        ; PAGE
***********************************************************
*     Section Name: copyfile                              *
*                                                         *
*        Function: Copies input file to output file       *
*                                                         *
*      Entry Data: d0 - file handle for (opened) output file *
*                  d1 - file handle for (opened) input file *
*                                                         *
*       Exit Data: d0 - return code (0 = success)         *
*                                                         *
***********************************************************

copyfile:
        movem.l d3-d6/a2-a4,-(a7)   ;save registers
        move.l  d0,d4               ;save output file handle
        move.l  d1,d5               ;save input file handle
        move.l  databuf(a5),d0      ;address of data buffer
        bne.s   cpfl010             ;bypass alloc if buffer exists
*-----------------------------------------------------------*
*       Allocate work buffer for reading/writing            *
*-----------------------------------------------------------*
        move.l  #RW_BUFS,d0     ;size of buffer
        move.l  #MEMF_CLEAR,d1  ;allocation type
        move.l  Absbase,a6      ;Exec base
*  *ptr = AllocMem(size,type)
*   d0              d0   d1
        CALL    AllocMem(a6)    ;allocate buffer
        move.l  d0,databuf(a5)  ;save and test address
        bne     cpfl010         ;continue if alloc successful
*-----------------------------------------------------------*
*       Unable to allocate buffer - gotta quit              *
*-----------------------------------------------------------*
        move.l  stdout(a5),d1   ;Std Output file handle
        lea     nomem(pc),a0
        move.l  a0,d2           ;address message
        move.l  #nomeml,d3      ;length of it.
        move.l  dosaddr(a5),a6  ;DOSBase into a6
*   Write(file,data,length)
*          d1   d2    d3
        CALL    Write(a6)       ;send message to standard out
        bra     cpfl_err        ;and leave

cpfl010:
*-----------------------------------------------------------*
*               Read input file                             *
*-----------------------------------------------------------*
        move.l  d5,d1           ;input file handle
        move.l  databuf(a5),d2  ;address of buffer
        move.l  #RW_BUFS,d3     ;max length to read
        move.l  dosaddr(a5),a6
*  actualLength = Read(file,buffer,maxLength)
*      d0              d1   d2      d3
        CALL    Read(a6)
        move.l  d0,d6           ;save actual length read
        beq     cpfl090         ;0 ==> end-of-file
        bpl.s   cpfl020         ; > 0 data read OK
```

```
*-----------------------------------------------------------------*
*       A Read error has occured                                  *
*-----------------------------------------------------------------*
        move.l  dosaddr(a5),a6
*  error = IoErr()
*    d0
        CALL    IoErr(a6)       ;get actual error value
        move.l  d0,work(a5)     ;store in parm list for Printf
        lea     rderrm(pc),a0   ;string pattern
        lea     work(a5),a1     ;arg array
        move.l  arpbase(a5),a6
*  count = Printf(*string,*arg-array)
*   d0              a0      a1
        CALL    Printf(a6)      ;write error message
        bra.s   cpfl_err        ;take error exit

cpfl020:
*-----------------------------------------------------------------*
*       Write same data to output file                            *
*-----------------------------------------------------------------*
        move.l  d4,d1           ;output file handle
        move.l  databuf(a5),d2  ;buffer address
        move.l  d6,d3           ;length of data read
        move.l  dosaddr(a5),a6
*  returnedLength = Write(file,buffer,length);
*      d0                  d1   d2      d3
        CALL    Write(a6)       ;write data to output file
        cmp.l   d0,d6           ;Q. did we write all the data?
        beq.s   cpfl010         ;A. YES - loop on for rest of data

*-----------------------------------------------------------------*
*  An error occurred writing, since not all data was written      *
*-----------------------------------------------------------------*
        move.l  dosaddr(a5),a6
*  error = IoErr()
*    d0
        CALL    IoErr(a6)       ;get actual error value
        move.l  d0,work(a5)     ;store in parm list for Printf
        lea     wrerrm(pc),a0   ;string pattern
        lea     work(a5),a1     ;arg array
        move.l  arpbase(a5),a6
*  count = Printf(*string,*arg-array)
*    d0              a0      a1
        CALL    Printf(a6)      ;write error message

cpfl_err:
        movem.l (a7)+,d3-d6/a2-a4 ;restore registers
        moveq.l #20,d0            ;error return code
        rts                       ;return to caller

cpfl090:
        movem.l (a7)+,d3-d6/a2-a4 ;restore registers
        clr.l   d0                ;good return code
        rts                       ;return

        ; PAGE
*****************************************************************
*     Section Name: getfib                                      *
*                                                               *
*        Function: Gets the FileInfoBlock for the passed        *
*                  named file                                   *
*                                                               *
*       Entry Data: a5 - Address of global work area            *
*                   a0 - address of file name string            *
*                                                               *
*        Exit Data: d0 - return code (0 = success)              *
*                   fib(a5) - filled in File Info block         *
*                                                               *
*****************************************************************

getfib:
        movem.l d2/d3,-(a7) ;save registers

*-----------------------------------------------------------------*
*        Get a shared lock on the file                            *
*-----------------------------------------------------------------*
        move.l  a0,d1           ;file name in d1
        move.l  #ACCESS_READ,d2 ;lock type
        move.l  dosaddr(a5),a6  ;DOSBase
*  lock = Lock(name,mode)
*   d0          d1   d2
        CALL    Lock(a6)
        tst.l   d0              ;Q.  Lock obtained?
        beq.s   gfib_err        ;A.  No - return error

        move.l  d0,d3           ;save lock
*-----------------------------------------------------------------*
*               now get File Info from lock                       *
*-----------------------------------------------------------------*
        move.l  d3,d1           ;lock in d1
        lea     fib(a5),a0      ;address fib
        move.l  a0,d2           ;FIB address in d2
        move.l  dosaddr(a5),a6
*  boolean = Examine(lock,&FileInfoBlock);
*    d0              d1       d2
        CALL    Examine(a6)
        move.l  d0,-(a7)        ;save Examine return code
*-----------------------------------------------------------------*
*       Free Lock (we are done with it)                           *
*-----------------------------------------------------------------*
        move.l  d3,d1           ;lock in d1
        move.l  dosaddr(a5),a6
*  UnLock(lock);
*         d1
        CALL    UnLock(a6)      ;free lock

        move.l  (a7)+,d0        ;restore return code from examine
        tst.l   d0              ;Q.  Good return from Examine?
        beq.s   gfib_err        ;A.  no - take error exit

*-----------------------------------------------------------------*
*               Successful processing                             *
*-----------------------------------------------------------------*
        clr.l   d0              ;set good return code
        movem.l (a7)+,d2/d3     ;restore registers
        rts                     ;and return

*-----------------------------------------------------------------*
*               Error exit                                        *
*-----------------------------------------------------------------*
gfib_err:
        moveq.l #20,d0          ;set bad return code
        movem.l (a7)+,d2/d3     ;restore registers
        rts                     ;and return

        ; PAGE
*****************************************************************
*     Section Name: setdate                                     *
*                                                               *
*        Function: Sets the datestamp for the output file       *
*                  to match that contained in the input FIB     *
*                                                               *
*       Entry Data: outbuf(a5) contains full path name of       *
*                       output file                             *
*                   a0 - the address of the input file's FIB    *
*                                                               *
*        Exit Data: N/A (Note: errors are considered            *
*                   non-critical and are ignored)               *
*****************************************************************

setdate:
        movem.l d3/d4/a3/a4,-(a7) ;save registers
        move.l  a0,a3             ;save address of fib
*-----------------------------------------------------------------*
*       Get Handler Process address                               *
*-----------------------------------------------------------------*
        lea     outbuf(a5),a0   ;get output file name
        move.l  a0,d1
        move.l  dosaddr(a5),a6
```

```
*  process - DeviceProc(name)
*     d0                d1
        CALL    DeviceProc(a6)
        tst.l   d0              ;Q. Handler found?
        beq.s   sdat090         ;A.  no - can't continue
        move.l  d0,a4           ;    yes - save handler address

*-----------------------------------------------------------------*
*       Get a lock on the output file                             *
*-----------------------------------------------------------------*

        lea     outbuf(a5),a0   ;name of output file
        move.l  a0,d1
        move.l  #ACCESS_READ,d2 ;lock type (shared)
        move.l  dosaddr(a5),a6
*  lock = Lock(name,accessMode)
*   d0          d1      d2
        CALL    Lock(a6)
        tst.l   d0              ;Q. Lock obtained?
        beq.s   sdat090         ;A. No - cannot continue
        move.l  d0,-(a7)        ;   yes - save lock on stack
*-----------------------------------------------------------------*
*               Get lock for parent directory                     *
*-----------------------------------------------------------------*
        move.l  d0,d1           ;lock into d1
*  lock = ParentDir(lock)
*   d0              d1
        CALL    ParentDir(a6)
        move.l  d0,d3           ;save lock (note: this CAN be zero)
*-----------------------------------------------------------------*
*  Free the first lock we obtained - don't need it any more       *
*-----------------------------------------------------------------*
        move.l  (a7)+,d1        ;lock for output file
*  UnLock(lock)
*         d1
        CALL    UnLock(a6)      ;free this lock
*-----------------------------------------------------------------*
*  Now build the file name as a BCPL string                       *
*       NOTE: this name MUST NOT include any path information,    *
*             just the file name                                  *
*-----------------------------------------------------------------*
        lea     strwork+1(a5),a0    ;string work area (saving the 1st byte)
        lea     fib_FileName(a3),a1 ;file name in passed FIB
        clr.l   d0                  ;set count to 0
1$:     addq.b  #1,d0               ;increment count
        move.b  (a1)+,(a0)+         ;move byte
        bne.s   1$                  ;loop until null byte
        subq.b  #1,d0               ;don't count null byte
        move.b  d0,strwork(a5)      ;length into first byte
*-----------------------------------------------------------------*
*       Build Argument list for DOS Packet                        *
*-----------------------------------------------------------------*
        clr.l   pktarg0(a5)         ;first argument null
        move.l  d3,pktarg1(a5)      ;second argument is lock of parent dir
        lea     strwork(a5),a0      ;pointer to BSTR file name
        move.l  a0,d0
        lsr.l   #2,d0               ;convert to BCPL pointer (BPTR)
        move.l  d0,pktarg2(a5)      ;third argument is BSTR ptr to file name
        lea     fib_DateStamp(a3),a0
        move.l  a0,pktarg3(a5)      ;fourth arg is REAL pointer to DateStamp
*-----------------------------------------------------------------*
*   Now use the ARP routine to send the DOS Packet                *
*-----------------------------------------------------------------*
        move.l  #ACTION_SET_DATE,d0
        lea     pktarg0(a5),a0
        move.l  a4,a1               ;handler process
        move.l  arpbase(a5),a6
*  Result1 = SendPacket(action, args, handler)
*    d0                  d0      a0      a1
        CALL    SendPacket(a6)
*-----------------------------------------------------------------*
* Ignore return code - if it worked, it worked - if not, not      *
*                                                                 *
*       Free the lock on the Parent Directory                     *
*-----------------------------------------------------------------*
        move.l  d3,d1
        move.l  dosaddr(a5),a6
*  UnLock(lock)
*         d1
        CALL    UnLock(a6)

sdat090:
        movem.l (a7)+,d3/d4/a3/a4       ;restore registers
        rts

        ; PAGE
*-----------------------------------------------------------------*
*                   Program Constants                             *
*-----------------------------------------------------------------*

dosname dc.b    'dos.library',0
arpname dc.b    'arp.library',0

        CNOP    0,4
nomem   dc.b    'Unable to allocate work area - aborted',$0a
nomeml  EQU     *-nomem

noarp   dc.b    'Unable to Open ARP Library - aborted',$0a
noarpl  EQU     *-noarp

nosinm  dc.b    'Unable to locate SINCE file - aborted',$0a
nosinml EQU     *-nosinm

noout   dc.b    'Unable to locate TO directory - aborted',$0a
nooutl  EQU     *-noout

notdir  dc.b    'Specified TO parameter is not a directory - aborted',$0a
notdirl EQU     *-notdir

templ   dc.b    'FROM/A,,,,,,,,,,,TO/A/K,SINCE/K',0

cmdpr   dc.b    'Usage:  ',$9b,'1;32mUpdate',$9b,'0;31m [FROM] ',$9b,'1;32mobj1 '
        dc.b    $9b,'0;31m[obj2 ... obj12] ',$9b,'1;32mTO dir',$9b
        dc.b    '0;31m [SINCE mfile]',$0a,$0a
        dc.b    '   Where ',$9b,'1mobj1...obj12',$9b,'0m are either files '
        dc.b    'or directories (with or without',$0a
        dc.b    '                wildcards, specifying input files to be checked'
        dc.b    $0a,$0a
        dc.b    '         ',$9b,'1mdir',$9b,'0m is the output directory '
        dc.b    'to receive the qualifying files',$0a,$0a
        dc.b    '    and  ',$9b,'1mmfile',$9b,'0m specifies the '
        dc.b    'optional file whose datestamp will be ',$0a
        dc.b    '                used to determine which input files '
        dc.b    'will updated in the ',$0a
        dc.b    '                output directory - NOTE that if no SINCE '
        dc.b    'file is specified,',$0a
        dc.b    '                the corresponding file in the TO directory '
        dc.b    'will be checked',$0a,0

ferrm   dc.b    'Command format error',$0a
ferrml  EQU     *-ferrm

fline   dc.b    '... %s ',0
foklin  dc.b    '  copied',$0a
foklinl EQU     *-foklin
iopnerm dc.b    '  Unable to Open Input file - Aborted',$0a
iopnerl EQU     *-iopnerm
oopnerm dc.b    '  Unable to Open Output file - Aborted',$0a
oopnerl EQU     *-oopnerm
exerm   dc.b    '  Error examining output file after locking - Aborted',$0a
exerl   EQU     *-exerm
rderrm  dc.b    ' Read Error - code %ld',$0a,0
wrerrm  dc.b    ' Write Error - code %ld',$0a,0
dlokerr dc.b    '  Unable to Lock Input directory %s - Aborted',$0a,0
dexerr  dc.b    '  Unable to Examine Input directory %s - Aborted',$0a,0
baddir  dc.b    '  Corrupted Directory chain - error %ld - Aborted',$0a,0

        ; PAGE

        END
```

AssemPro

Review by Chris Zamara

Assembly language development system
From Abacus, 5370 52nd Street SE, Grand Rapids, MI, 49508
Suggested retail $99.95 (US)

AssemPro is an assembly language development system for the Amiga. It is an integrated environment that consists of an editor, assembler, debugger, and a help utility that gives instant access to information about assembler op-codes and system library routines. These four elements of AssemPro are implemented as independent windows, each knowing certain things about what's going on in the others.

The main strength that AssemPro has over other assemblers is the convenience afforded by its interactive, integrated development environment. For example, you can assemble a program directly from the editor; errors are reported as they are encountered via a requester that lets you correct the error on the spot and try again. Alternatively, you can have an error file generated that the editor can read, which lets you browse through your source by choosing "next error" to find out where problems lie. What you avoid is the cycle of changing your source, saving it, then compiling (and linking) separately. Another example of how the editor ties in with the rest of the system: move the cursor to an op-code and hit help, and a window pops up and shows the available addressing modes.

You may think that with all of this on-line help, AssemPro would be ideally suited to beginners in Assembler. To a certain extent this is true, and in fact the program may be of limited interest to the serious software developer because of its major flaw: it has no linker. If you stop to consider the implications of this shortcoming, you may conclude that you won't want to use AssemPro for major assembly language programming projects. On the other hand, if you are learning assembly or want to put together relatively small programs, AssemPro's no-hassle nature can be a boon. We consider this package in the same class of assemblers as *PAL* on the Commodore 64, which are used for their virtues of simplicity of use and friendliness.

The basic design behind the system seems to be a fine way to make your way through the code/assemble/test/debug development cycle. First of all, the text editor is surprisingly good. It has all the usual menu-driven functions (although range operations behave in a non-standard way), and has a more powerful underlying command-driven syntax as well. This lets casual users edit without having to learn anything, while catering to the power user at the same time. Command strings can be bound to function keys, letting you define your own text processing operations and access them easily.

The editor has other nice touches as well, like the proportional gadgets that move the text up and down or left and right dynamically as you move them, and the fact that you can edit multiple documents at a time, each in its own window. The best reason to use this editor for writing assembler source code, however, is its language-sensitivity. For example, shift-cursor up or down brings you to the previous or next label in your source code; the search function has an option that lets you search for symbols only; you can read an error file produced by the assembler and skip to the next or previous error in your source. This latter feature is especially useful, because you traverse your source code, not the error file, and when you skip to an offending line, the error message is displayed in the editor status window. That's the kind of convenience that you can get only in an integrated editor/assembler system such as this.

When it comes time to assemble your source, you can go to the Assembler window and select "Assemble" from a menu, or just double-click right in the editor to bring up the assembly requester (although the latter method is not documented!). If you haven't chosen the "error file" option from the assembler, errors are presented one at a time by displaying the bad line of source in a string gadget within a requester. At this stage you have the option of cancelling the assembly, or correcting the line (which will change it in your source in the editor) and trying again. Once again, this integrated approach is much quicker than re-editing, saving and asssembling after every error.

The assembler itself is fairly standard in terms of the mnemonics it uses, syntax, etc. It supports macros, conditional assembly, interactive assembly-time user input, local variables, and just about everything you would expect in a 68000 assembler. To partially compensate for the fact that there is no linker, there is a provision for saving and loading symbol files, which is sort of like exporting and importing symbols between modules. All symbols between an "SLABEL" and "ENDS" pseudo-op pair will be saved to the designated file; another module giving the "ILABEL" command with that filename would then have access to the other module's macros and constants, since it would know about all symbols in that module. This capability can save assembly time by not having to re-compile header files, etc., but you still can't develop a large program in pieces the way you can when using a linker.

The next level beyond editing and assembling is, of course, debugging, and, while not as extensive as, for example, Manx's DB or Metadigm's Metascope debuggers, the AssemPro debugger gives you an easy way to see your code and data as it actually sits in memory, set breakpoints, and execute or single step through your code. The debugger isn't integrated

with the assembler the way the editor is, so you have to save a file from the editor and load it into the debugger.

Once a file is loaded into the debugger, you see a disassembly of the code, and you can scroll up and down through memory, examining code and data as you go. The debugger doesn't distinguish between code and data hunks, though, so you have to recognize your data and switch from disassembly to hex dump modes yourself. If you chose the "save symbols" option from the assembler, and choose "relocate symbols" from the debugger, your variable names and code entry points will be clearly marked so that you can follow along with the code. You can test buggy code in relative safety, since the debugger will trap most errors before they "software error" or "Guru" the system. You can search through memory for certain data, and single step through every instruction, or just those at the current level, so that you don't enter subroutines. There is even a "68020 single-step" emulation mode, which only stops on branching, subroutine, and TRAP instructions. The debugger also is also a convenient way to look at values at known memory addresses, or to disassemble code anywhere in memory (like ROM routines!).

You can also access a Disassembler and "Reassembler" from the debugger window. The disassembler will disassemble code from memory and output a nicely formatted listing to printer or to any file. The Reassembler is similar, but produces a file which can be assembled again.

The fourth main part of the package is the "Tables" window, which lets you find the available addressing modes for any 68000 instruction. You can also load tables for all of the system routines and find the register calling conventions for any routine. This is a useful part of the package, especially for beginners to 68000 assembler and/or Amiga programming.

The four windows come up in the four quarters of the screen, but can be sized and moved as desired (you can save your favourite set-up to a configuration file). The operation of the windows is not consistent; for example, clicking the close gadget on the assembler or debugger windows exits AssemPro, while clicking close on the editor window erases the current document, and closing the Tables window just makes the window its original size and pushes it to the back. The four windows are not separate tasks, so when one is busy it holds up the others - a bad implementation for a program on the Amiga. It would be nice to be able to edit or at least look through a source file during a long assembly, for example.

As good as the whole set-up sounds in theory, trying to actually do something with it can be difficult. For example, we originally found it impossible to assemble using the standard system include files (which are supplied with AssemPro), because the INCLUDE directive always looks in the current directory for its files. Specifying the entire path to the include file you want isn't a solution, because most of the system include files include other files, which the assembler will then fail to find. We finally got this to work by changing the current directory to the include directory, but then AssemPro will look there every time you go to load or save a file of any kind. This is an unbelievable oversight on the part of AssemPro's designers, but it is understandable: they obviously never use the standard include files. The example programs that come on the disk define everything they need at the top of the file; every structure member reference, every library routine offset, every flag and constant. This is definitely *not* the way to go about programming on the Amiga, and it makes things especially difficult for the beginner, who has no way to know what he needs to define at the top of his file. Even if you do, going through the include files to pick out everything you need from them is far too tedious. Once you save common symbol definitions with the "ILABEL" command, you will be set for most of your programming, but it would have been nice if these files had come on the disk and the manual encouraged you to use them.

Which brings us to the manual, which is excellent in quality as far as typesetting and binding, but somewhat weak in its content. It is obviously written by programmers, since it gets into a lot of complicated detail at times, and leaves out the big picture, i.e. how the application works and feels from a user's perspective. It uses the Macintosh terms for Amiga entities, calling Drawers "folders", and Requesters "dialog boxes" throughout. The program was written in Germany (the comments in some of the sample programs are in German!), and it looks as if the manual has been poorly translated. Spelling mistakes abound. In some cases the manual doesn't reflect the program accurately, as some features of the program aren't documented and names of menu options are sometimes different in the manual than in the program. To its credit, the manual is quite complete, has useful reference tables for commands, and covers the menu options and features of each part of the program thoroughly.

There are also a few small problems that can cause difficulties: the editor and assembler have fixed data buffer sizes, so you can run out of memory editing or assembling a large program, even if there is plenty of memory left in the system (you then have to increase the memory size with a "memory" menu option); if you try to move a window right after sizing it, the system will often hang up - this bug is mentioned in the manual, so it was recognized, but never fixed; the editor always saves the current file when you exit, even if it hasn't been changed.

As a whole, AssemPro evokes some mixed feelings. On one hand, its basic design and feel are excellent, but it has enough inconsistencies, bugs, oversights and oddities that it can be infuriating - especially so when you realize the potential that is there. The general impression one gets is that the program was designed with the best of intentions, but rushed out the door, resulting in a rough and incomplete implementation. This, of course, is better than a program designed poorly in the first place, since it can be fixed. Abacus has assured us, in fact, that they fully intend to support this program with upgrades, so the bugs and problems that get in your way may clear out and leave you with the powerful and convenient system that AssemPro is supposed to be.

Photon Paint

A Step into the Future of Amiga Graphics

Review by Louis Markoya

Louis Markoya is a Senior Engineer for a semiconductor equipment company. His fine art experience includes some one-man shows and a five-year apprenticeship with Spanish Surrealist Salvador Dali'. He's been working with microcomputers since 1980 and finds that the Amiga provides the artist with a vehicle to excel in computer graphics without the need for expensive, unavailable systems. He can be contacted on CompuServe at 75766,504, on any Fido network board which carries the Video echo, or at 49 Walnut Avenue, Shelton, Connecticut 06484.

Photon Paint $99.95
from MicroIllusions, 17408 Chatsworth St., Granada Hills, CA 91344. 800-522-2041

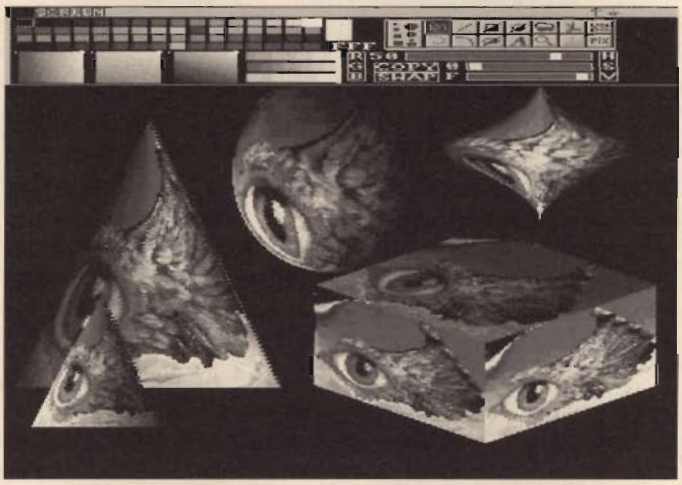
Since the release of Commodore's Graphicraft, Amiga paint programs have evolved significantly to offer state of the art features and give artists, illustrators and electronic doodlers of all ages the tools to produce outstanding graphics on a micro. Offering shading, dithering, perspective, special brush manipulation, and anti-aliasing, Deluxe Paint II was the first of the second generation paint programs to be offered for the Amiga. The next significant program to appear was Digi-Paint, the HAM (Hold and Modify) paint program from NewTek. It offered, for the first time, the ability to use HAM mode, and the 4096 colours of the Amiga effectively. Among its advanced features are an amazing array of painting modes, with smooth blending, transparencies, logical functions and more.

With the release of Photon Paint, MicroIllusions has created a third generation graphics program, which not only combines the outstanding features of both Deluxe Paint II and Digi-Paint, but adds features not available on *any* micro, and found only in the most sophisticated, dedicated high-end graphics systems.

Photon Paint is a full-featured HAM mode paint system which, through the use of many palette remapping options and a large array of blending and shading modes, enhances the power of HAM mode and the Amiga's 4096 colours to their fullest. Among the outstanding features are real-time operation of freehand drawing tools, the ability to paint in exactly the chosen colour (without HAM artifacts), highly sophisticated brush functions including twisting, variable transparency, and mouse-operated perspective tilting, 3D wrapping of brushes on solid geometric shapes as well as free-drawn symmetric shapes with an adjustable light source for shading, and a relocatable magnification window with variable magnification and scroll gadgets.

Now let's launch into the program to discuss these and other features in more detail. Photon Paint comes on two disks, a program disk and a picture disk which contains tutorial images. The program disk includes the Workbench and can be booted from the familiar hand/disk prompt. The program will operate on 512k systems but only in medium-res HAM mode using no interlace or overscan. To use these features will require a megabyte. For the full power of the program, utilizing double screen buffering and extra screen size, more than a meg is recommended. (Photon Paint manages memory quite efficiently, allowing double buffering in a one meg machine, but as more memory is used for the screens, tool features are dropped by the program.) After double clicking the Photon icon you are greeted by a smooth scrolling start-up screen. The scrolling is also used for all of the programs requesters, making them unique and attention grabbing. After loading the program, a black screen appears with a relocatable tool bar at the top called the "Fast Menu". The Fast Menu can be toggled from either a pull down menu or the keyboard. Indeed, most Photon Paint features have hot key equivalents. The Fast Menu includes the current palette consisting of 64 redefinable colours, rectangles containing the current foreground and background colours, and a complete array of painting tools.

The tools offered are familiar to most computer artists but extra features have been built in. A brush selection area is provided offering eight predefined brushes, a freehand drawing tool, a dotted line freehand drawing tool, a line tool (rubber banding), a freeform arc tool, a hollow or filled rectangle tool, a hollow or filled ellipse tool, a hollow or filled circle tool, a text tool, a fill tool, a magnification tool, a scissors/lasso tool, a restore previous brush tool, an undo tool and the PIX, or pixelization tool. On the left of the Fast Menu appears the word "SCREEN". By clicking on this gadget, the Photon Paint screen can be pulled down and the Workbench or other application will be revealed. Photon is very well behaved and multitasks nicely. (You'll need more than 1 meg if you plan to multitask in interlace and/or overscan mode). On the right side of the Fast Menu are two arrows. Click on these and the Fast Menu is expanded to include a complete set of palette tools, consisting of three HAM colour arrays, RGB (red, green and blue) sliders, HSV (Hue, Saturation and Value) sliders and copy and swap tools. Using this menu one can also create colour spreads in the palette.



Three types of fill operations are supported by the program. 'Freeform fill' will fill any closed shape you draw with the current selected colour. 'Background fill' fills any screen area of the same colour up to another, or boundary colour. The third type is the 'boundary fill': You first select the boundary colour to fill to, and then the area inside that boundary. This area will be filled to the boundary colour with the current colour selection regardless of colour change.

Selecting the Magnification tool brings up a unique magnifier window (see photo on previous page). This window is resizable and relocatable, and offers gadgets for scrolling and variable magnification.

Two types of cut operations are supported in the Scissors and Lasso tools. Selecting the Scissors tool will allow you to drag a box around the area you wish to pick up as a brush, while double clicking this tool will turn it into a Lasso, which will pick up any part of your picture by drawing around it freeform. The background colour can be specified as transparent for either of these modes, allowing you to pick up only the object you want and nothing around it. Brushes cut this way or imported from another source are called "Pic brushes".

The PIX tool is unique - it lets you 'pixelize' areas, creating a mosaic, computer-generated look. The size of the pixel squares is adjustable from 1 to 9, allowing for blends and varying degrees of the effect.

The real drawing power of Photon Paint is that all the above drawing and fill operations, even Pic brushes, can be used in any of its drawing modes including Normal, Blend (with extraordinary setting capabilities), Add and Subtract. These will be explained further in the description of the menus.

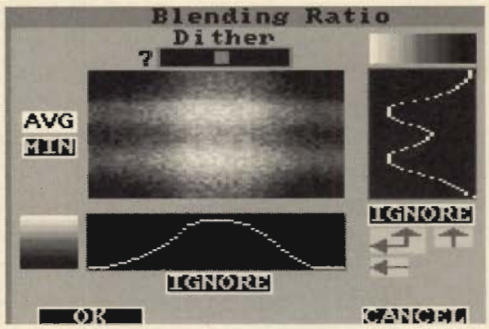
Holding down the right mouse button will display Photon Paint's menu selections: Project, Prefs, Brush, Fonts and Mode. The Project menu contains the usual items, as well as some extra ones: The Fast Menu selection will toggle the Fast Menu (as will either the Help or F10 key). The next item, Base Colours, allows you to choose or remap the HAM colour palette from your brush, picture, palette or alternate picture. This will allow you to use HAM to its greatest potential, and avoid artifacting. The Alternate selection will allow you to switch screens to a second buffered screen (simply 'j' from the keyboard) or delete what is in either screen. This "extra" work space is great for preparing parts of your painting while not affecting the work you've already done.

Within the Preferences menu are all the choices which form the conditions you will paint under. In Screen Mode you can choose to work in interlace (320x400) or overscan (352x240) by clicking on their gadgets. Selecting Exact Colour will allow you to draw in the exact colour chosen at any time, instead of having your colour translated by HAM into something you never expected. Also available here are selections for coordinate display, Mouse Speed, Mouse Following and Screen Offsets. Selecting Mouse Follow will assure that the exact path you draw will be followed, even if you are drawing faster than the program can calculate the HAM colours needed for your line. The last item, Save Prefs, allows you to save the selections you have made so they will be activated the next time you start the program.

The Brush menu is where Photon Paint excels and outshines other paint programs. All the brush operations work with Pic Brushes, which are any area you pick up with the scissors or lasso, or any image saved as a brush. Under this menu you will find the Load, Save As, Trans (transparent background), Flip, Resize, Twist, Rotate, Wrap On, Bend, Tilt, Lum, Remap and Free options. The Resize option allows for Doubling and Halving the Horizontal and/or Vertical axis as well as a dynamic resize. Rotate will allow + and - 90 degree rotations, 180's and a free rotate that will dynamically rotate to any angle. The Bend function bends the current brush in any direction you move the mouse.

The Tilt option places your 2D brush in 3D perspective on the

screen. Pitch (tilt up or down), Roll (pivot about a vertical centerline), Yaw (rotate the brush about its own center), Distance (front to back) and dragging the brush along a vanishing point are all made possible under this selection. A wireframe outline changes dynamically to show the effect of dragging the mouse. Since any mouse movement translates to one of the effects, you are provided with a numeric readout of Pitch, Tilt and Yaw to make things easier, and the ability to lock out movement in any chosen axis. In addition, the specific effect desired can be chosen by activating its numeric key.



The Twist selection will, as its name suggests, twist your brush like taffy. Once again a wireframe dynamically shows the effect of the mouse movement. The further you move the mouse the more twists, or spirals, in the brush. Once you are satisfied with the outline, a tap on the spacebar will start the remap of the image.

The most powerful brush function is Wrap On. This option will allow you to wrap the image of your brush on a 3D solid object. The object choices are Tube, Cone, Ball, Ellipse, Free (you draw the right hand side of an object that will be mirrored), Cube3/4 (wraps your brush around any 3 visible sides of a cube) and Cube*3 which draws your brush on each visible side of the cube. These options are made most dramatic by using them with the Lum option. This option allows for the placement of a light source to add highlights and smoothly shade the the object your brush is being mapped on. By selecting Lum you are presented with a requester that allows you to set the light direction, intensity, amount and type of illumination (backlit or not). Only ray tracers have offered such power before, and of them, few offer IFF mapping.

Remap will remap your current brush to the picture's palette, allowing for the brush's actual colours to still be used. Selecting Free will clear the memory used for the last Pic brush.

The fourth choice in the main menu bar is the Fonts menu. Under this menu can be found the Style (Italic, Bold, Underline, Outline and Reverse), List Fonts, and Change Disk (which allows for changing font directories) choices. The program will only allow for the listing of fifteen fonts in any one directory, so multiple directories are the best way to manage this small shortcoming.

The last menu choice is Mode. The selections here affect the mode in which any of the drawing or filling functions operate, using normal or pic brushes. The choices are Normal, and Blend with its submenus Set, Add and Subtract. Normal allows straight unmodified paint and fill functions, while Add will add the current colour to the Background it is painted on. Likewise Subtract subtracts the brush colour from the background it is applied over. The real strength in the Mode menu is found in the Blend option. It allows the use of several outstanding features which allow Amiga artists to free their imagination and experiment endlessly. When the Blend option is chosen, the actual painting mode is determined through the Set parameter. Here, Blending ratios, transparency, amount of dithering and directional highlights can be chosen through a unique requester that allows for the balance of transparency to be drawn freehand in both the horizontal and vertical axes (see photo). This looks like a waveform editor for a sound program but instead of changing amplitude you are actually shaping where your brush or filled object will be most opaque, and how it will blend with the background. Within the requester is found a dither level control, horizontal and vertical transparency level indicators, editing windows for horizontal and vertical shading, a visual representation of your settings, and various copy and swap waveform gadgets. These and other tools in this selector make Photon Paint more powerful than any paint program for blending functions.

Photon Paint's manual is complete, easy to read and brings you through all of the program's functions in a logical order. The tutorial section and images are especially helpful in becoming accustomed to the program's use and features.

In this review I have tried to describe all of Photon Paint's strengths. The program has been a pleasure to use and I highly recommend it. The only drawbacks I found were no support of colour fonts, no polygon/filled polygon tool and no 'Again' tool (which is so helpful in Digi-Paint). However, I have been informed that these features and more (airbrush, grids, multicolour dithering, dropshadows and folding brushes) are being developed for a future release (not expected until late summer). In closing, I feel that MicrolIlusions has given the Amiga artist even more of an edge - the cutting edge of paint program technology and a step into the future for Amiga graphics.

* * * * * *

As a last minute bulletin, I have learned that RGB Video Creations, the "Delux Help" people, will be offering a "Delux Help for Photon Paint". Delux Help products are real-time tutorials covering virtually every feature of the software, the use of which will make a "power user" out of anyone in no time. The Photon Paint Help should be out at the end of March. The forerunners to this program are useful and powerful and, although it was not ready to test at the time of this review, I feel the program will be a great help to all of those who would like to take full advantage of Photon Paint's power, without spending weeks to learn it.

RGB Video Creations can be reached at 3944 Florida Blvd., Suite 102, Palm Beach Gardens, Florida 33410.

News LINK

Transactor News

Dbase Blues: It's taken quite a toll in terms of time, effort and crises but at last the new subscriber database is up and running. The mailing of the premiere issue showed up a few mistakes, but these are now, we hope, a thing of the past. Please check your label very carefully and let us know of any changes or corrections as soon as possible. If you have a subscription to both *Transactor for the Amiga* and *Transactor* you will now have one subscriber number for both; any other orders will also be recorded under this number. Now, if you have a query, we should be able to answer it right away and cut down dramatically on errors.

Customer Satisfaction: Now, to ensure that you receive a speedy response to your queries and orders, Renanne Turner is heading up Customer Service. Just drop her a line here in Richmond Hill or phone during office hours and she'll be happy to help you.

Assistant Editor: Malcolm O'Brien has joined the editorial team and has been responsible for producing the label text for the 128 Fish Disks currently available from *Transactor for the Amiga*. This was an enormous task but we think you'll agree it was worthwhile when you see the result. Malcolm has written in the past for *Transactor* and other magazines and we are very happy to welcome him aboard.

Fish Disks: We are now shipping these and apologize for the delay. Each disk has its contents listed on the label; disk #0 is a catalogue of all the disks currently available and is continuously updated. Fish Disks are produced by Fred Fish and are packed full of public domain and freely redistributable programs including games, utilities, music, graphics and just about anything else you can think of; the Workbench icons are worth the price alone. Complete sets of labels for Fish Disks are available at $25 per set. We regret that individual labels are not available.

***Transactor for the Amiga* Disk #2:** This has been held up and we apologize for the delay. However, it will be available by the time you read this and should be worth the wait.

Volume 1 Issue 1 Disk: Oh, oh! Yes there was a bug, in the Cycle Warriors program. The fix is given on page 80; it is also on Disk V1 I2, and on many on-line services as well. Disks sent out from this office with the proper pink labels have been fixed. We apologize for any frustration and will endeavour to ensure it doesn't happen again. Disk subscribers please note that you will receive a properly typeset label for your V1 I1 disk along with your V1 I2 disk.

Just a reminder that the disk *accompanies* the magazine, it is not a magazine on a disk. The disk contains all the programs, source code, utilities and so forth from the magazine together with as many "extras" as we can pack onto it. Each disk has a properly typeset label, coloured to match the relevant magazine, giving a complete directory of its contents. Due to many requests and suggestions, the disk for Volume 1 Issue 2 will operate from Workbench Icons as well as from the CLI.

UEDIT V2.3: Undoubtedly one of the best text editing tools available for the Amiga. V2.3 has a new emphasis on Word Processing, while preserving UEdit's power and versatility. Serial numbers are assigned to registered users, who are then eligible for the newsletter and updates regularly produced by the author, Rick Stiles. UEdit can be customized right from the startup or at any other time without a necessity for recompiling the entire config. Fully documented on the disk, this is the editor to have running from its tiny window whenever you're using your machine. Price and ordering details are on the card in the centre of the magazine.

Industry News

SupraDrive Hard Disk for Amiga 500: Capable of burst data transfer of over 250KB per second. Available in 20, 30, 60 and 250MB capacities at $995, $1195, $1995 and $3995 respectively, it comes ready to plug in and use. For information, contact John Wiley at Supra Corporation, 1133 Commercial Way, Albany, OR, 97321 (503) 967-9075.

G.O.M.F. 2.0 - The Guru-Buster: Hypertek/Silicon Springs Development Corp. announce G.O.M.F. 2.0, an error handling routine which effectively eliminates Guru Meditation Alert messages by allowing the user to end the program causing the error and save his data without having to reset the computer.

GOMF (Get Outa My Face) was originally released as shareware; the new, enhanced version traps and removes *all* errors during the running of virtually any program(s), including the direct GURU alert. It automatically configures itself to work on 68000/68010/68020 and offers messages in plain English describing what it is doing and prompting the user to perform the required steps. Complete with detailed instructions, GOMF 2.0 retails at $34.95 US/$46.95 CDN.

Also from Hypertek comes the **Deluxe MIDI Interface** for all Amigas. Compatible with all programs that use serial MIDI standard output it offers serial port pass-thru, MIDI IN, MIDI THRU and a switchable THRU/OUT; has over six feet of cable and a one year warranty. Please specify model when ordering.

The TTL High Resolution Monitor for the Amiga plugs into RGB port for crisp, flicker-free, high-resolution monochrome video output. Included is a disk with a special Workbench font, an RGB port pass-through and a one-year warranty. Amiga 2000 and sidecar owners may use this monitor in PC HiRes mode.

The Amiga Light Pen is designed to work with any Amiga program. Its software allows the use of both the mouse and light pen in precision graphics and features single-pixel precision, variable sensitivity, zoom mode, button toggle, etc. It comes with Inkwell Systems' DT-184 2-button light pen and a pen-hook.

For information on all these products contact Hypertek at 120-1140 Austin Avenue, Coquitlam, BC, Canada, V3K 3P5 (604) 939-8235.

Sound Quest Inc. has introduced two new Editor/Librarian programs for the Yamaha DX711 and the Roland D50. Utilizing the Amiga, both products have been designed for both the professional and serious amateur musician.

The DX11 Master Editor/Librarian provides "musician friendly" help screens for each Editor window; loads data files from disk into the Amiga or directly to the synthesizer singly or in batches; and loads stored system-exclusive data to any appropriate MIDI instrument. Multi-tasking is utilized to permit the editing of as many data banks as desired - limited only by the computer's memory - and eight bank editing functions may be performed on any sized group of patches at one time. The DX11 stores and edits all 13 types of DX711 Sys Ex data and adds fractional scaling, micro-tuning, performance, set-up and personally controlled random voice generation.

The D50 Master Editor/Librarian provides an identically helpful environment. Editing is performed on partials, tones or patches from a main window and features pop-up windows for graphic envelope editing. It can 'lock' any combination of partials or tones for simultaneous editing. For more information contact Glenn Hayworth, Sound Quest Inc., 5 Glenaden Avenue East, Toronto, Canada, M8Y 2L2 (416)234-0347

Comspec Autobooting Hard Drive for the Amiga 1000:

• Auto Booting - Kickstart and Workbench are automatically booted from the hard disk when you power up. There is no need to swap disks even after a 'Guru Meditation'.

• Handles media defects transparently to the user.

• Intelligent SCSI controller chip reduces CPU overhead during disk access and allows high speed multitasking even when the drive is being used.

• Hard drive chassis with enough room to add hard drives from 10 - 300 MB or space to add a second half height hard drive or tape streamer. Complete with a power supply capable of running from 100 VAC to 240 VAC at 50Hz or 60Hz without setting any switches or jumpers and a low noise fan to protect from overheating.

For information contact Comspec Communications Inc., 153 Bridgeland Avenue, Unit 5, Toronto, ONT, Canada, M6A 2Y6. (416) 785-3553 Fax: (416) 785-3668.

Business Products from Brown-Wagh Publishing:

The Works by Micro-Systems software combines the Analyze! spreadsheet, Scribble! word processor and Organize! database in one package - ideal as a business starter-kit at $199.95 US.

Publisher Plus by Northeast Software Group is an upgrade of Publisher 1000 with PostScript laser support and additional functions at $99.95 US.

TV*SHOW by Zuma Group helps the Amiga owner put together a slick slide show or live video presentation and have fun doing it at $99.95 US.

Write & File by Softwood Company represents the newest generation of WYSIWYG word processors for the Amiga, with multiple font styles onscreen, spell checker, mail merge and an integrated database manager at $99.95 US.

For information contact Brown-Wagh Publishing, 16795 Lark Avenue, Suite 210, Los Gatos, CA, 95030, (408) 395-3838.

New from Progressive Peripherals & Software:

EXP-1000 and 512 RAM expansion series for the Amiga 500 featuring easy installation, new features and options. Prices from $169.95 US.

Prodrive External 3.5 inch disk drive for the Amiga 500, 1000 and 2000 and the Prodrive 2000 Internal 3.5 inch disk drive for the Amiga 2000. ($239.95 US.)

PIXmate professional image enhancement package enables you to modify individual colour values including hue, saturation and luminance simultaneously; convert 4096 colour HAM images to 32 colour and back; use two independent screens for processing images and for "cut and paste" between screens; "grab" Deluxe Paint screens for titling or retouching before importing the image into PIXmate; make a colour negative of current image and convert it to positive without darkroom processing and numerous other options. Price $69.95 US.

IntroCAD entry level CAD software specifically designed for the Amiga. It uses the Amiga's intuitive user interface with professional CAD features, customized printer drivers and

many features and options. Price $79.95 US.

Also available now: TermPro; Access 64; Mathamation; Diskmaster; Framegrabber; EXP-8000+; UltraCAD; Spectrum and ProGEN. For further information contact Erich Stein, Progressive Peripherals & Software, 464 Kalamath Street, Denver, CO, USA, 80204 (303) 825-4144

Keeping Fit with Bodylog Inc.: Fitness packages designed to run on the Amiga ranging in price from $59.95 to $269.95 including the HeartLab program, which uses Bodylog's Sensor-BASIC program that lets you program the sensors in BASIC. The prices include the Pulse Rate Sensor and the Sensor Adapter Cable. Designed for people concerned with their cardio-fitness and for those wanting to create their own applications, the package is very easy to use. Create your own experiments to see what is happening inside your own body. Further information from Jerald Greenberg, Bodylog Inc., 34 Maple Avenue, Armonk, New York 10504. Tel (914)273-6480.

Hotlicks and Galileo from Infinity Software: Hotlicks brings the sound of music to the non-musician. It features Amiga's four channels of digitized sound, standard SMUS scores and 8SVX digitized instruments. All the scores, instruments and sound effects are original, created just for Hotlicks.

Galileo is an astronomy program that takes full advantage of the Amiga's high resolution graphics and processor speed. You can scroll rapidly to any part of the sky; develop constellation identification; calculate rise and set times and plot pathways. The Yale Bright Star Catalogue disk gives supplemental data with an additional 9100 stars. It was developed by Mike Smithwick, an astronomer and computer graphics artist.

Galileo and Hotlicks are available from Infinity Software, 1144, 65th Street, Suite C, Emeryville, CA, 94608. (415)420-1551 Fax: (415) 420-1729.

3 For 2 Amiga 500 Offer for Canadian Schools: Commodore is offering three A500 computers for the price of two to boards of education, high schools, elementary schools, community colleges and universities across Canada until July 31st. The system includes a 1084 high-res monitor, an integrated 94-key keyboard with a separate numeric keypad, 512K of RAM (expandable to 1MB), one 3" disk drive, an optomechanical mouse and a variety of connectors and ports for printers and other peripherals. For information, contact CBM Ltd., 3470 Pharmacy Avenue, Agincourt, Ont. M1W 3G3. Phone (416) 499-4292.

Abacus Books and Software: *Amiga for Beginners, Amiga Tricks and Tips, Amiga Machine Language, Amiga BASIC Inside & Out* and *Unix/Xenix Reference Guide* are new reference books from Abacus. The **TextPro** wordprocessor, **BeckerText** 'professional' word processor, **DataRetrieve** database manager, and **AssemPro** assembler are also now available. For further information and prices contact Rob Lun, Abacus, 5370, 52nd Street SE, Grand Rapids, MI, 49508. (616)698-0330.

Tshell for the Amiga from Metran Technology. Provides power and versatility for software development. Simple to learn and use, it is far more advanced than the CLI and its alternatives, being more like an interpreted programming language. It is an improvement on standard Unix's shells, and was designed especially for the Amiga from the bottom up.

Features include a Unix-like interface, C-Like syntax, command line editing and history, online documentation via the HELP key, 40 built-in commands and many, many more.

Introductory Special Offer Price is $50 from Metran Technology and this includes a free update which will be strongly influenced by feedback from Tshell users. Metran Technology, Box 890, West Oneonta, NY, 13861.

MCP Associates announce the Grand Opening of their Amiga-oriented retail store in Queens, New York. They stock a full range of Amigas including hardware, peripherals, software, disks and accessories. For information phone William Poll or Christine Ladikos at (718) 956-9000 or visit the store at 28-21 Astoria Blvd., Astoria, New York, 11102. Present a copy of *Transactor for the Amiga* when you make your purchase and you will receive a **gift.**

A 6 ft. Amiga 1000 to Thomson 4120 RGB Monitor Cable - Code: I-94. Available from Interex, 717 South Emporia, Wichita, KS, USA, 67211-2307. (316) 264-6118.

The Wedge XT-controller interface from RSI of British Columbia is a two-piece interface that plugs into the side of an Amiga 1000. The Wedge accepts an IBM PC XT-style controller card in its extra slot, which connects directly to a variety of low-cost hard drives. The Wedge comes complete with associated software and online support from the RSI's Mind Link. An A500 Wedge will be available in the near future.

Suggested retail price is $159 US ($195 Cdn). For information, contact RSI, c/o 9651 Alexandra Road, Richmond, BC Canada, V6X 1C6, (604) 270-0064.

Forth for the Amiga: From Delta Research comes JForth, an Amiga specific interpreter and compiler in one language providing an interactive environment enabling you to compile programs directly from the keyboard and debug them immediately. The system has a number of features to completely integrate it with the Amiga environment. The manual has an extensive JForth glossary, detailed appendices and a tutorial for

first time users. Available at $99.95 US from Delta Research, 4054 Wilkie Way, Palo Alto, CA., 94306. (415) 856-3669.

Compiler Companion from Lattice: Featuring EXTRACT and BUILD, which extract file names from a directory and build a command file from the list; CXREF, which generates a cross-reference listing of C language source files; DIFF, which compares files and reports the differences; FILES, which locates files by specified attributes; GREP, which matches character string patterns within a file; LMK, the Lattice Make Utility, which automates the management of files based on dependencies and rules; SPLAT, which searches and replaces within one or more files; TOUCH, which updates the date and time stamps of individual files; and WC, which counts the number of words within a file. The price is $99.00; Compiler Companion is available through dealers and distributors worldwide or directly from Lattice, Inc., 2500 South Highland Avenue, Lombard, IL, 60148. (312) 916-1600.

Hurricane Accelerator Board: Hurricane combines a Motorola 68020 microprocessor and 68881 math coprocessor on a board, replacing the 68000 microprocessor in the Amiga. These components make your Amiga up to 40 times faster than an IBM AT and up to 5 times faster than a VAX 11/780 while retaining full software compatibility. Hurricane speeds up applications even without memory expansion and can be easily installed. An optional memory board containing up to 4 megabytes of 32-bit RAM can be connected to the Hurricane for even faster performance. It works in conjunction with any memory you may already have. The Hurricane board is $1045 (A1000); $1095 (A2000); the 32-bit memory expansion board is $495 (0k-unpopulated); $895 (2Mb); $1295 (4Mb) from Finally Technologies, 25 Van Ness, San Francisco, CA, 94102. (415) 564-5903.

Dual 3.5" Disk Drives from Comp-U-Save: 100% compatible with all Amiga computers, the drives have their own internal power supply and are colour coordinated with the Amiga. Features include an all metal chassis, horizontal layout, vented enclosure, on-off switch, 6 foot 3-prong linecord and primary circuit breaker protection. The price is $395. There is also a 20 Meg hard drive (SCSI) with controller at $785. Information from Comp-U-Save, 414 Maple Avenue, Westbury, NY 11590. (516)997-6707 in NY or outside call 1-800-356-9997.

Amiga 500 Command Centre: KEYTEK's Command Centre places two 3.5" external disk drives (or one 3.5" drive and a 20 or 30 Mb hard disk drive) just above and directly behind the keyboard for easy access. It features a built-in cooling fan, surge suppression, EMI noise filter and a five-outlet power strip with a convenient bank of remote switches for fingertip control of individual system components. Optional accessories include a 20 or 30 MB hard disk drive, a 1 or 2 Mb internal RAM upgrade, a heavy duty replacement power supply, joystick extension cables, a dust cover and a tilt/swivel monitor stand. The price is $179.95 or $99.95 for the standard cabinet without the electrical components. Available from KETEK, Inc., PO Box 203, Oakdale, IA, 52319. (800) 626-4582.

Oops!

A problem in the Warrior Cycles program in Volume 1 Issue 1 caused the system to report a false "Read Write Error" when 'cycle' was run from a floppy disk. The bug actually lies with the system, and occurs when a value of zero is posted to the timer device. Before any players have entered the game, Warrior Cycles tells the timer device to wait zero microseconds at a time. This seems to wreak havoc with the TrackDisk device, and disk access from floppy just isn't the same after that.

If you are equipped to compile, the fix is easy. At around line 102 of the file "cycles.c", you'll find the following code:

```
if ((TIMEREQ *)msg == tr) { /* time request reply */
  move_cycles(&cycle_list);
  Timer_Post(tr,
    TIME*num_alive/1000,              /* seconds */
    ((num_alive*TIME)%1000)*1000  /* micros  */
            );
```

You can see that when 'num_alive' is zero (which is the case when the game starts, before any players have joined), Timer_Post() will get zeroes for its seconds and microseconds arguments. Change this to read:

```
if ((TIMEREQ *)msg == tr) { /* time request reply */
  if (num_alive) {
    move_cycles(&cycle_list);
    Timer_Post(tr,
      TIME*num_alive/1000,            /* seconds */
      ((num_alive*TIME)%1000)*1000 /* micros */
            );
  }
  else
    Timer_Post(tr, TIME/1000, (TIME%1000)*1000);
```

This gives Timer_Post() nonzero time values in the case where 'num_alive' is zero. The Transactor disks being shipped have the fixed version on them.

Although the bug was annoying, it helped pinpoint the system bug to the timer device; previous reports mentioned that the problem was with the DOS Delay() and WaitForChar() routines.

for the AMIGA Transactor

The Magazine for Amiga Programmers

Please print your name and address below (or transfer your mailing label)
Check here for change of address ☐

Name: _____

Address: _____

City: _____ *Province/State:* _____

Zip/Postal: _____ *Phone:* _____ *CompuServe ID:* _____

Subscribers only: please enter subscription number from mailing label ☐

Subscription Order Section

	Canada ($Cdn.)			U.S.A. ($US)			Foreign ($US)			Air Mail ($US, overseas only)		
	Mag	Disk*	Both*	Mag	Disk	Both	Mag	Disk	Both	Mag	Disk	Both
1 year	☐ 19.00	☐ 67.00	☐ 86.00	☐ 15.00	☐ 55.00	☐ 70.00	☐ 21.00	☐ 65.00	☐ 86.00	☐ 40.00	☐ 79.00	☐ 119.00
2 years	☐ 35.00	☐ 123.00	☐ 158.00	☐ 27.50	☐ 101.00	☐ 128.50	☐ 38.50	☐ 119.50	☐ 157.50	☐ 73.00	☐ 145.00	☐ 218.00
3 years	☐ 47.50	☐ 167.50	☐ 215.00	☐ 37.50	☐ 137.50	☐ 175.00	☐ 52.50	☐ 162.50	☐ 215.00	☐ 100.00	☐ 197.50	☐ 297.50

*** Ontario residents please add 7% of disk subscription price for provincial sales tax. No PST on magazine subscriptions.**

☐ **Please send Microfiche instead of magazines** ☐ **Send me info on Transactor Classic (C64/128)**

Combo Subscription Bonus!

If you are subscribing to BOTH the magazine and the disk, check one of the following FREE gifts:

☐ **Transactor Classic Amiga Disk #1** ☐ **Fish Catalog Disk** ☐ **Extended Subscription (7 Issues)**

Product Order Section

Ontario residents please add 7% PST on all items

$Cdn.	$U.S.	
☐ 14.95	12.95	Transactor Classic Amiga Disk #1
☐ 14.95	12.95	Transactor Classic Amiga Disk #2
☐ 58.00	45.00	Uedit Text Editor, v2.3f

Fish Disks $6.95 US, $7.95 Cdn. each

Note: Disk numbers not yet released may be ordered in advance - simply add numbers to end of list. Disks 57, 80 and 88 are not available.

☐ 1	☐ 11	☐ 21	☐ 31	☐ 41	☐ 51	☐ 62	☐ 72	☐ 83	☐ 94	☐ 104	☐ 114	☐ 124
☐ 2	☐ 12	☐ 22	☐ 32	☐ 42	☐ 52	☐ 63	☐ 73	☐ 84	☐ 95	☐ 105	☐ 115	☐ 125
☐ 3	☐ 13	☐ 23	☐ 33	☐ 43	☐ 53	☐ 64	☐ 74	☐ 85	☐ 96	☐ 106	☐ 116	☐ 126
☐ 4	☐ 14	☐ 24	☐ 34	☐ 44	☐ 54	☐ 65	☐ 75	☐ 86	☐ 97	☐ 107	☐ 117	☐ 127
☐ 5	☐ 15	☐ 25	☐ 35	☐ 45	☐ 55	☐ 66	☐ 76	☐ 87	☐ 98	☐ 108	☐ 118	☐ 128
☐ 6	☐ 16	☐ 26	☐ 36	☐ 46	☐ 56	☐ 67	☐ 77	☐ 89	☐ 99	☐ 109	☐ 119	☐ 129
☐ 7	☐ 17	☐ 27	☐ 37	☐ 47	☐ 58	☐ 68	☐ 78	☐ 90	☐ 100	☐ 110	☐ 120	☐ 130
☐ 8	☐ 18	☐ 28	☐ 38	☐ 48	☐ 59	☐ 69	☐ 79	☐ 91	☐ 101	☐ 111	☐ 121	☐ 131
☐ 9	☐ 19	☐ 29	☐ 39	☐ 49	☐ 60	☐ 70	☐ 81	☐ 92	☐ 102	☐ 112	☐ 122	☐ 132
☐ 10	☐ 20	☐ 30	☐ 40	☐ 50	☐ 61	☐ 71	☐ 82	☐ 93	☐ 103	☐ 113	☐ 123	☐ 133

The Transactor for the *AMIGA* Disk $12.95 US, $14.95 Cdn. each
(these disks correspond to issues of **Transactor for the *AMIGA***)

☐ Vol. 1, Iss. 1 ☐ Vol. 1, Iss. 2

Add flat $2.00 shipping & handling for any number of items ordered from above box.

Transactor for the *AMIGA* Back Issues and Microfiche: $4.50 U.S., $4.50 Cdn. (MF = Microfiche)

☐ Vol. 1, Iss. 1 ☐ MF ☐ Vol. 1, Iss. 2 ☐ MF

Ontario residents please add 7% provincial sales tax for any items ordered from this box.

☐ Visa ☐ MasterCard ☐ Cheque/MO enclosed Cheque #_____ Dated ____/____/____ Amount _____

Acct. # _____ Expires ____/____ Signature _____

NOTE: Prepayment required. Purchase orders will be accepted ONLY if accompanied by payment.

Please allow 6-8 weeks for delivery in the U.S., 4-6 weeks in Canada.

FOLD ALONG SCORES WITH CORRECT REPLY CARD ON OUTSIDE. USE TAPE, PLEASE - NO STAPLES OR GLUE.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 390 BUFFALO, NY

POSTAGE WILL BE PAID BY ADDRESSEE

The Transactor
P.O. Box 338 Station 'C'
Buffalo, NY, 14209-9990

Business Reply Mail
No Postage Stamp Necessary
if mailed in Canada

Postage will be paid by:

Canada Post Postes Canada
46487

The Transactor
501 Alden Road
P.O. Box 3250
Markham Industrial Park
Markham, Ontario
L3R 9Z9

Reader Service Card

Fill out the card (please print) or affix your peel-off mailing label. This card valid until June 30, 1988.

Name _____

Company _____

Address _____

_____

City _____ *Prov/State* _____

Postal/Zip Code _____

Phone# _____

See Advertiser Index
for corresponding service numbers

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110
111	112	113	114	115	116	117	118	119	120
121	122	123	124	125	126	127	128	129	130
131	132	133	134	135	136	137	138	139	140
141	142	143	144	145	146	147	148	149	150
151	152	153	154	155	156	157	158	159	160
161	162	163	164	165	166	167	168	169	170
171	172	173	174	175	176	177	178	179	180
181	182	183	184	185	186	187	188	189	190
191	192	193	194	195	196	197	198	199	200

Advertiser Index

Advertiser	Page	Reader Service #
Abacus Software	Inside front cover	1
ASDG Inc.	49	2
Free Spirit Software, Inc.	37	3
Interface Technologies Corporation	1	4
MCP Associates, Inc.	Inside back cover	5
MicroIllusions	Outside back cover	6
RSN Software, Inc.	1	7
Syndesis	23	8

TransAmi Fish Disk Labels

Since we're selling the Fred Fish collection of public domain programs, we decided to make useful labels for each of the disks. Each program on a disk is displayed prominently on the label and, where there is room, a one-line description of each program is given next to the name. Below are samples of two labels. All Fish disks you order from us are labelled, and if you already have the disks, you can order the entire set of labels (currently up to disk #128) for just $25.00 US.

Fish Disk #60

From Fred Fish's Library of Freely Redistributable Amiga Software

Blitz	Fast text file viewer
BlitzFonts	Text output up to 6 times faster
HandShake	VT52/VT100/full VT102 emulation V1.20a
Med	V2.1 - mouse-oriented editor
PrtDrvGen	Generate custom printer drivers - V1.1
Show	Slideshow program - V2.1
Uedit	V2.0 shareware editor -configure/customize
Ueturbo	Customized Uedit for development environment

Fish Disk #56

From Fred Fish's Library of Freely Redistributable Amiga Software

ClipBoard	Clipboard device interface routines
ConPackets	Demos the use of DOS packets
GetDisks	Find all available disk device names
GetVolume	Get volume name for a given file
Icon2C	Icon file -> icon data structures
MergeMem	Allocate memory which spans board boundaries
mCAD	Object-oriented drawing, V 1.1

place
stamp
here

Transactor
85-10 West Wilmot Street
Richmond Hill, Ontario
L4B 1K7

Coming up in future issues...

Sadly, once again we were left with too much material for the allotted 80 pages for this issue, and we had to delay two excellent articles for the next one. One of these was a detailed explanation of how ray tracing works, right down to the mathematical theory and C code needed to do the job. The article is by Cathryn Graham, author of the documentation for the programs Sculpt-3D and Animate 3D, which were written by her husband Eric Graham. The article is packed with explanation, mathematical theory, diagrams, and short C functions. The second article that had to be postponed was about the proposed ANSI standard for C compilers, written by Eric Giguere. This should clear up some of the confusion about the standard, which is a hot topic in the compiler world right now.

The cover art originally scheduled for this issue was also held off for next time so that we could tie in Louis Markoya's picture with his review of Photon Paint. The picture we were going to use - and the one you'll see next time - is by Jo-Anne Park, a well-known Amiga artist from Toronto. Her cover piece is a real eye-grabber (although not quite as *literally* as Markoya's!), and in quite a different vein from both previous covers.

Also scheduled for a future issue is an article by John Toebes on internationalizing your programs so that they'll work properly for Amiga users in all countries. Also look for an article on the BridgeBoard, and more from some well known writers and programmers in the Amiga community.

THE DISK SPECIALISTS

Disk Supplies

Bulk

C.ITOH, Fujitsu

DSDD – Blue

50 –	$1.25
100 – 350	1.20

DSDD – Rainbow

50 –	$1.49
100 – 350	1.45

SSDD - BLUE

50 -	.99¢
100 - 350	.97¢

Bulk

SONY

DSDD – Blue

50 –	$1.29
100 – 350	1.29

M C P

Associates, Inc.
DISTRIBUTORS

featuring

C.ITOH, Fujitsu Kasei, SONY, etc:

3.5" Double Sided, Double & High Density
100% Certified – Lifetime Warranty!

Bulk – 50 disks per carton, priced per disk
Boxed – 10 " " box, priced per box.

(we honour our manufacturers' warranties)

Special Offer: AMUSE PD Disks

5 disks for $30.00

Write for listing – limited time offer. By arrangement. AMUSE – Sponsors, AmiEXPO!

Labels, Packaging

Boxed

C.ITOH, Fujitsu

DSDD – Blue

1 – 3	$15.95
4 – 9	15.45

DSDD – Rainbow*

1 – 3	$23.95
4 – 9	22.95

* includes plastic case

SONY

DSHD – High Density*

1 – 3	$55.95
4 – 9	53.95

* limited availability

Shipping/Handling Info: Minimum order: Bulk – 50 disks: Box – 2 boxes. We ship worldwide.
U.S. – Minimum Fee: $ 5.00, plus $3.50 per 100 disks bulk, or 2 boxes. For CODs, add $5.00
Canada – " " 13.00, " 5.50 " " " " " " . No CODs to Canada.
Prices subject to change and availability. Charges do not reflect any customs/duties.
MASTERCARD/VISA accepted: minimum purchase – $50.00. New York residents add sales tax.

Please note: Other brands, formats available. Specify brand, type. Disk Supplies, Labels. **Write/Call** for complete listings, and for Quantity Disk Pricing Schedule, & Specials.

Authorized AMIGA Dealer/Education Dealer.
Systems, hardware, software, peripherals.
Corporate and Educational Accounts welcomed.
Dealer inquiries invited.

Exceptional Service and Quality

M C P Associates, Inc., P.O.Box 6260, Dept. TA, Long Island City, New York, 11106-0260

(718) 956–9000

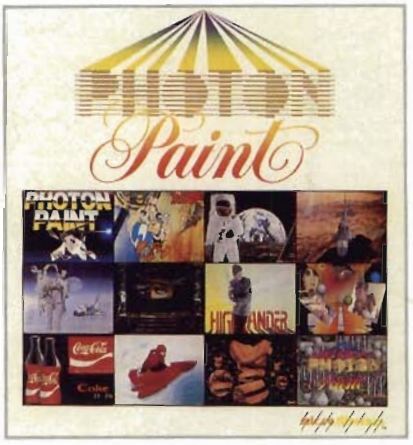
PHOTON Paint

EXPANDING YOUR PAINT CAPABILITIES AS NEVER BEFORE

PHOTON PAINT IS A POWERFUL LO- AND HI-RESOLUTION HOLD AND MODIFY (HAM) PAINT PROGRAM, WITH OVERSCAN (NTSC /PAL) AND SPECIAL EFFECTS FOR THE AMIGA COMPUTER!

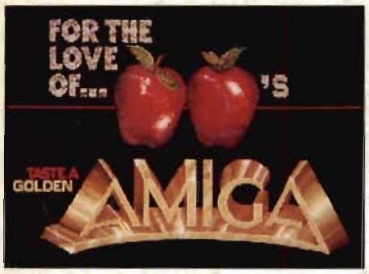
Now many of the features and techniques previously only available in separate Amiga paint programs have been combined and expanded in PHOTON PAINT! These include features that were limited to either a 32 color paint program, or a HAM paint program. Yet it doesn't stop there! PHOTON PAINT expands your paint capabilities as never before with these features:

- 64 color palette with 4,096 color alterations
- State of the art Surface Mapping
- Exceptionally sophisticated brush operations



- Real-time operation of drawing tools
- Paint in exactly the color you choose in HAM mode
- Allow HAM to modify colors
- Two types of fill functions
- All popular brush operations
- Full blend mode
- Total menu movability with brush and color palette menu size reduction option
- Light source specification, normal, intense or dithered
- Unique magnification window with variable pixel sizing
- Transparent or opaque background colors
- Palette color spreads with both (RGB) AND (HSV) functions
- Compatible with most third party art and animation systems, including Photon Video.

EXPAND YOUR CREATIVITY TO A NEW DIMENSION, PICK UP PHOTON PAINT TODAY!



microillusions™

17408 Chatsworth St., Granada Hills, CA 91344
Inside CA 818/360-3715 • Outside CA 800/522-2041 • FAX 818/360-1464